Part 1: Introduction to Transformers and NumPy Setup

1.1 What is a Transformer?

The Transformer is a novel neural network architecture introduced in the paper "Attention Is All You Need" by Vaswani et al. (2017). It has revolutionized the field of Natural Language Processing (NLP) and has found applications in various other domains like computer vision and reinforcement learning.

Unlike traditional sequence-to-sequence models like Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTMs) networks, Transformers do not rely on recurrence. Instead, they process entire sequences of data at once using a mechanism called **attention**, specifically **self-attention**. This allows for significant parallelization during training and enables the model to capture long-range dependencies in the data more effectively.

Key characteristics of Transformers:

- **Self-Attention Mechanisms:** These allow the model to weigh the importance of different parts of the input sequence when processing a particular element.
- **Parallel Processing:** Unlike RNNs that process tokens sequentially, Transformers can process all tokens in a sequence simultaneously.
- **Encoder-Decoder Structure:** The original Transformer consists of an encoder to process the input sequence and a decoder to generate the output sequence.
- **Positional Encoding:** Since Transformers don't have an inherent notion of sequence order, positional information is explicitly added to the input embeddings.

Transformers have become the foundation for many state-of-the-art models like BERT, GPT, and T5.

1.2 Why NumPy?

While deep learning frameworks like TensorFlow and PyTorch provide high-level abstractions and automatic differentiation, implementing a Transformer from scratch using only NumPy offers several benefits:

- **Fundamental Understanding:** It forces a deeper understanding of the underlying mathematics and mechanics of each component.
- **Clarity:** Without the black boxes of high-level libraries, the flow of data and computations becomes more transparent.
- **Educational Value:** It's an excellent exercise for learning how these complex models are built from basic building blocks.
- **Lightweight:** NumPy is a fundamental package for numerical computation in Python and has minimal overhead.

Our goal here is not to build the most optimized Transformer, but to understand its core components intimately.

1.3 Setting Up the Environment

To follow this tutorial, you'll need Python and NumPy installed.

Installation:

If you don't have NumPy installed, you can install it using pip:

```
pip install numpy
```

Verification:

You can verify the installation by opening a Python interpreter and typing:

```
import numpy as np
print(np.__version__)
```

This should print the installed version of NumPy.

1.4 Basic NumPy Operations for Transformers

We'll be using several NumPy operations extensively. Here's a quick refresher on some of the most important ones:

- **Array Creation:**

```python
# Create a 1D array
arr1d = np.array([1, 2, 3])
print("1D Array:\n", arr1d)

# Create a 2D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", arr2d)

# Create an array of zeros
zeros_arr = np.zeros((2, 3)) # Shape (2 rows, 3 columns)
print("Zeros Array:\n", zeros_arr)

# Create an array of ones
ones_arr = np.ones((3, 2))
print("Ones Array:\n", ones_arr)

# Create an array with random values (uniform distribution between 0 and 1)
rand_arr = np.random.rand(2, 2)
print("Random Array (uniform):\n", rand_arr)

# Create an array with random values (standard normal distribution)
randn_arr = np.random.randn(2, 2)
print("Random Array (normal):\n", randn_arr)
```

- **Array Attributes:**

```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Shape:", arr.shape)    # (2, 3)
print("Dimensions:", arr.ndim) # 2
print("Data type:", arr.dtype) # int64 (or similar, depending on your system)
print("Size (total elements):", arr.size) # 6
```

- **Matrix Multiplication (Dot Product):**
  This is fundamental to many operations in Transformers, especially attention.

```python
mat_a = np.array([[1, 2], [3, 4]])
mat_b = np.array([[5, 6], [7, 8]])

# Element-wise multiplication
# print("Element-wise product:\n", mat_a * mat_b)

# Matrix multiplication (dot product)
dot_product = np.dot(mat_a, mat_b)
# Alternatively, using the @ operator (Python 3.5+)
# dot_product_alt = mat_a @ mat_b
print("Dot Product:\n", dot_product)
# Expected output:
# [[1*5+2*7, 1*6+2*8],
#  [3*5+4*7, 3*6+4*8]]
# = [[19, 22],
#    [43, 50]]
```

- **Transpose:**

```python
mat = np.array([[1, 2, 3], [4, 5, 6]])
print("Original Matrix:\n", mat)
print("Transposed Matrix:\n", mat.T)
# Or np.transpose(mat)
```

- **Summation, Mean, Max, Min:**

```python
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("Sum of all elements:", np.sum(arr))
print("Sum along axis 0 (columns):", np.sum(arr, axis=0))
print("Sum along axis 1 (rows):", np.sum(arr, axis=1))

print("Mean of all elements:", np.mean(arr))
print("Max element:", np.max(arr))
print("Min element:", np.min(arr))
```

- **Broadcasting:**
  NumPy's broadcasting allows operations on arrays of different shapes under certain constraints. This is very powerful.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 10
print("Array + scalar:\n", arr + scalar) # Scalar added to each element

row_vector = np.array([10, 20, 30])
print("Array + row_vector:\n", arr + row_vector) # Row vector added to each row

col_vector = np.array([[10], [20]])
print("Array + col_vector:\n", arr + col_vector) # Column vector added to each column
```

- **Indexing and Slicing:**

```
arr = np.array([0, 10, 20, 30, 40, 50])
print("Element at index 2:", arr[2]) # 20
print("Elements from index 1 to 3 (exclusive):", arr[1:3]) # [10, 20]
print("All elements from index 2:", arr[2:]) # [20, 30, 40, 50]

mat = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Element at row 0, col 1:", mat[0, 1]) # 2
print("First row:", mat[0, :]) # [1, 2, 3]
print("First column:", mat[:, 0]) # [1, 4, 7]
print("Submatrix (rows 0-1, cols 1-2):\n", mat[0:2, 1:3])
# [[2, 3],
#  [5, 6]]
```

- **Mathematical Functions:**
  NumPy provides a wide range of mathematical functions that operate element-wise on arrays.

```python
arr = np.array([1, 2, 3])
print("Square root:", np.sqrt(arr))
print("Exponential:", np.exp(arr))
print("Logarithm (natural):", np.log(arr))

# Softmax (we will implement this later, but good to know np.exp and np.sum)
def simple_softmax(x):
    e_x = np.exp(x - np.max(x)) # Subtract max for numerical stability
    return e_x / e_x.sum(axis=-1, keepdims=True)

scores = np.array([1.0, 2.0, 0.5])
print("Simple Softmax:", simple_softmax(scores))
```

1.5 What's Next?

In the next part, we will dive into the core mechanism of the Transformer: **Scaled Dot-Product Attention**. We'll understand its mathematical formulation and implement it using NumPy.

Stay tuned!

Part 2: Implementing Scaled Dot-Product Attention

Welcome to Part 2 of our Transformer from scratch series! In this part, we'll implement the fundamental building block of the Transformer's attention mechanism: **Scaled Dot-Product Attention**.

2.1 What is Attention?

In the context of neural networks, particularly sequence processing, **attention** is a mechanism that allows a model to selectively focus on certain parts of an input sequence when producing an output. Instead of treating all parts of the input equally, the model learns to assign different "attention weights" to different input elements, signifying their relevance to the current processing step.

Think of it like how humans pay attention. When you read a sentence, you might focus more on certain words to understand its meaning. Attention mechanisms in neural networks try to mimic this behavior.

2.2 Scaled Dot-Product Attention

Scaled Dot-Product Attention is one of the most common and effective types of attention mechanisms, and it's the one used in the original Transformer paper.

It operates on three inputs:

- **Queries (Q):** A set of vectors representing what we are looking for.
- **Keys (K):** A set of vectors representing the information available in the input sequence.
- **Values (V):** A set of vectors representing the actual content or features of the input sequence.

Intuitive Explanation with an Example

To make Q, K, and V more concrete, let's use an example sentence: "I have a cat and i love her."

Imagine the model is processing this sentence and needs to understand the relationships between words, particularly what "her" refers to.

- **Query (Q):** Think of the Query as the current word or concept the model is focusing on and trying to gather more information about. It's like asking a question.
    - *Example:* If the model is trying to understand what "her" refers to, then "her" (or its vector representation) acts as the Query. The implicit "question" is "Who or what is 'her' referring to in this context?"
- **Key (K):** Think of Keys as labels or identifiers for all the words in the sentence. Each word in the input sequence has a Key associated with it. This Key is a vector that represents the word's content in a way that can be compared to the Query.
    - *Example:* Every word in "I have a cat and i love her" would have a corresponding Key vector. The Key for "cat" would be a vector representing the concept of "cat". Similarly

for "I", "have", etc.

- **Value (V):** Think of Values as the actual content, meaning, or rich representation of each word. Once the Query, by comparing itself to all Keys, identifies which words are most relevant, it uses their corresponding Values to get the information it needs.
    - *Example:* Each word also has a Value vector. The Value for "cat" is the rich semantic representation of "cat" that the model can use.

How it works with the example "I have a cat and i love her":

Let's say the model is processing the word "her" (this is our **Query**).

1. The model takes the Query vector for "her" and compares it against all the **Key** vectors in the sentence (i.e., the Key for "I", "have", "a", "cat", "and", "i", "love", "her"). This comparison is typically done using a dot product.
2. This comparison calculates a score for each Key. The score between the Query "her" and the Key "cat" will likely be high, because "her" (a pronoun) often refers to something mentioned earlier, and "cat" is a plausible antecedent in this context. The score between "her" and, say, "love" might be lower if the primary goal is to find the referent.
3. These scores are then passed through a softmax function, which converts them into "attention weights". A high weight for the Key "cat" means the Query "her" should pay a lot of attention to "cat". These weights sum to 1, representing a distribution of attention.
4. The model then takes these attention weights and uses them to compute a weighted sum of all the **Value** vectors. If the Key "cat" received a high attention weight, its corresponding Value vector contributes significantly to the resulting sum.
5. The output of this process is a new vector, a refined representation for "her". This new vector now incorporates contextual information, strongly influenced by the Value of "cat", helping the model understand that "her" refers to "cat".

In essence, for a given Query (what the model is currently focusing on):

- It uses **Keys** to determine "how relevant" every other part of the input is.
- It then uses the **Values** of the relevant parts (weighted by their relevance) to construct a richer, context-aware representation of the Query.

This mechanism allows the model to dynamically decide which parts of the input are most important for understanding or processing each specific part of the sequence.

The core idea is to compute a score for each key with respect to a given query. This score determines how much attention the query should pay to that particular key-value pair. The scores are then used to compute a weighted sum of the values, resulting in the output of the attention layer.

Mathematical Formula:

The formula for Scaled Dot-Product Attention is:

```
Attention(Q, K, V) = softmax( (Q @ K.T) / sqrt(d_k) ) @ V
```

Let's break down each part:

1. **`Q @ K.T` (Dot Product of Queries and Transposed Keys):**
   - `Q` has dimensions `(num_queries, d_k)` or `(batch_size, seq_len_q, d_k)` if batched.
   - `K` has dimensions `(num_keys, d_k)` or `(batch_size, seq_len_k, d_k)` if batched.
   - `K.T` (transpose of K) will have dimensions `(d_k, num_keys)` or `(batch_size, d_k, seq_len_k)`.
   - The dot product `Q @ K.T` results in a matrix of **attention scores** (also called compatibility scores or alignment scores) with dimensions `(num_queries, num_keys)` or `(batch_size, seq_len_q, seq_len_k)`. Each element `(i, j)` in this matrix represents how much query `i` aligns with key `j`.
2. **`sqrt(d_k)` (Scaling Factor):**
   - `d_k` is the dimension of the key vectors (and query vectors, as they must have the same dimension for the dot product).
   - The scores are scaled down by dividing by the square root of `d_k`. This scaling is crucial because for large values of `d_k`, the dot products can grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. Scaling helps to counteract this effect, leading to more stable training.
3. **`softmax(...)` (Softmax Function):**
   - The softmax function is applied row-wise to the scaled scores. This converts the scores into probabilities (non-negative values that sum to 1 across each row).
   - The output of the softmax, often called **attention weights**, will have the same dimensions as the scaled scores: `(num_queries, num_keys)` or `(batch_size, seq_len_q, seq_len_k)`.
   - Each row `i` of the attention weights represents the distribution of attention that query `i` pays to all the keys.

4. `... @ V` **(Dot Product with Values):**
    - The attention weights are then multiplied by the `V` (Values) matrix.
    - `V` has dimensions `(num_values, d_v)` or `(batch_size, seq_len_v, d_v)`. Note that `num_keys` must equal `num_values` (or `seq_len_k` must equal `seq_len_v`), as each key corresponds to a value.
    - The dimension `d_v` is the dimension of the value vectors. It can be different from `d_k`.
    - The final output of the attention mechanism has dimensions `(num_queries, d_v)` or `(batch_size, seq_len_q, d_v)`. Each output vector is a weighted sum of the value vectors, where the weights are determined by the attention scores.

Optional Masking:

In some scenarios (like in the decoder of a Transformer, or for handling padded sequences), we might want to prevent attention to certain positions. This is done by adding a **mask** (typically a large negative number like -infinity for elements we want to ignore) to the scaled scores *before* applying the softmax. The softmax will then assign near-zero probabilities to these masked positions.

2.3 NumPy Implementation

Let's implement the Scaled Dot-Product Attention using NumPy.

```
import numpy as np

def softmax(x, axis=-1):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True)) # Subtract max for numerical stability
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Calculate the attention weights and the output of the attention mechanism.

    Args:
        Q (np.ndarray): Query matrix, shape (..., seq_len_q, d_k)
        K (np.ndarray): Key matrix, shape (..., seq_len_k, d_k)
        V (np.ndarray): Value matrix, shape (..., seq_len_v, d_v) where seq_len_k == seq_len_v
        mask (np.ndarray, optional): Mask to apply to the attention scores, shape (..., seq_len_q,

    Returns:
        tuple: (output, attention_weights)
            - output (np.ndarray): The output of the attention mechanism, shape (..., seq_len_q, d
            - attention_weights (np.ndarray): The attention weights, shape (..., seq_len_q, seq_le
    """
    d_k = Q.shape[-1]  # Dimension of the key vectors

    # 1. Calculate dot product of Q and K.T: (..., seq_len_q, d_k) @ (..., d_k, seq_len_k) -> (...
    # We need to transpose the last two dimensions of K for matrix multiplication
    # K.transpose(0, 1, 3, 2) if K is (batch_size, num_heads, seq_len_k, d_k)
    # For simpler case (seq_len_k, d_k), K.T is fine.
    # For (batch_size, seq_len_k, d_k), K.transpose(0, 2, 1) is needed.
    # np.matmul handles broadcasting for batch dimensions correctly.
    scores = np.matmul(Q, K.swapaxes(-2, -1))  # Q @ K.T

    # 2. Scale the scores
    scaled_scores = scores / np.sqrt(d_k)

    # 3. Apply mask (if provided)
    if mask is not None:
        # The mask should be shaped to broadcast correctly with scaled_scores.
        # Typically, mask elements are 0 for positions to attend to and 1 for masked positions.
        # We want to add a large negative number to masked positions.
        scaled_scores = np.where(mask == 0, scaled_scores, -1e9) # or -np.inf for true infinity

    # 4. Apply softmax to get attention weights
    # Softmax is applied on the last axis (seq_len_k) to get probabilities over keys for each quer
    attention_weights = softmax(scaled_scores, axis=-1)

    # 5. Multiply attention weights by V: (..., seq_len_q, seq_len_k) @ (..., seq_len_v, d_v) -> (
    # Note: seq_len_k must be equal to seq_len_v (number of keys must be number of values)
    output = np.matmul(attention_weights, V)

    return output, attention_weights
```

Explanation of the Code:

1. **`softmax(x, axis=-1)` function:**
   - This is a standard softmax implementation.
   - `np.max(x, axis=axis, keepdims=True)` is subtracted from `x` before exponentiation. This is a common trick for numerical stability, preventing overflow when `x` contains large values. It doesn't change the output of softmax because `softmax(x) = softmax(x - c)` for any constant `c`.
   - `keepdims=True` ensures that the dimensions are preserved after max and sum operations, allowing for proper broadcasting.
   - `axis=-1` means the softmax is computed along the last axis of the input array.
2. **`scaled_dot_product_attention(Q, K, V, mask=None)` function:**
   - `d_k = Q.shape[-1]`: We get the dimension of the key vectors from the last dimension of `Q`.
   - `scores = np.matmul(Q, K.swapaxes(-2, -1))`: This computes `Q @ K.T`. `K.swapaxes(-2, -1)` transposes the last two dimensions of `K`, which is necessary for the matrix multiplication. `np.matmul` correctly handles batch dimensions if `Q` and `K` are 3D or higher (e.g., `(batch_size, seq_len, d_model)`).
   - `scaled_scores = scores / np.sqrt(d_k)`: Scales the scores.
   - **Masking:**
     - If a `mask` is provided, it's used to modify `scaled_scores`.
     - The `mask` is expected to have `0` where attention is allowed and `1` (or `True`) where it should be prevented.

    - `np.where(mask == 0, scaled_scores, -1e9)`: This line is crucial. If `mask` at a position is `0` (allow attention), it keeps the original `scaled_scores`. If `mask` is `1` (prevent attention), it replaces the score with a very large negative number (`-1e9`). When softmax is applied, these large negative numbers will result in near-zero probabilities.
- `attention_weights = softmax(scaled_scores, axis=-1)`: Computes the attention weights using our softmax function. The softmax is applied along the last axis (`seq_len_k`), so for each query, the weights for all keys sum to 1.
- `output = np.matmul(attention_weights, V)`: Computes the final output by taking the weighted sum of `V` using the `attention_weights`.
- The function returns both the final `output` and the `attention_weights` (which can be useful for visualization or analysis).

2.4 Simple Input/Output Examples

Let's test our implementation with some simple examples.

Example 1: Basic Attention (No Batching)

Suppose we have:

- A single query vector.
- Three key vectors.
- Three corresponding value vectors.

```
# Setup: Dimensions
seq_len_q = 1  # Number of queries (e.g., current word we are focusing on)
seq_len_k = 3  # Number of keys/values in the sequence (e.g., words in the source sentence)
d_k = 2        # Dimension of keys and queries
d_v = 4        # Dimension of values

# Initialize Q, K, V with some dummy data
np.random.seed(42) # for reproducibility
Q_ex1 = np.random.rand(seq_len_q, d_k)   # (1, 2)
K_ex1 = np.random.rand(seq_len_k, d_k)   # (3, 2)
V_ex1 = np.random.rand(seq_len_k, d_v)   # (3, 4)

print("Q_ex1:\n", Q_ex1)
print("K_ex1:\n", K_ex1)
print("V_ex1:\n", V_ex1)

# Calculate attention
output_ex1, attention_weights_ex1 = scaled_dot_product_attention(Q_ex1, K_ex1, V_ex1)

print("\n--- Example 1 Output ---")
print("Attention Weights (shape: {}):\n".format(attention_weights_ex1.shape), attention_weights_ex
print("Output (shape: {}):\n".format(output_ex1.shape), output_ex1)

# Expected shapes:
# Attention Weights: (1, 3) - one query, attention over 3 keys
# Output: (1, 4) - one query, output dimension d_v
```

Example 2: Batched Attention with Masking

Now, let's consider a batch of sequences and apply a mask.

```
# Setup: Dimensions for batch
batch_size = 2
seq_len_q_b = 2  # Number of queries per item in batch
seq_len_k_b = 3  # Number of keys/values per item in batch
d_k_b = 4        # Dimension of keys and queries
d_v_b = 5        # Dimension of values

# Initialize Q, K, V with some dummy data for batch
np.random.seed(123)
Q_ex2 = np.random.rand(batch_size, seq_len_q_b, d_k_b) # (2, 2, 4)
K_ex2 = np.random.rand(batch_size, seq_len_k_b, d_k_b) # (2, 3, 4)
V_ex2 = np.random.rand(batch_size, seq_len_k_b, d_v_b) # (2, 3, 5)

# Create a mask
# Suppose for the first item in batch, the second query cannot attend to the third key.
# And for the second item, the first query cannot attend to the first key.
# Mask: 0 = attend, 1 = do not attend
mask_ex2 = np.zeros((batch_size, seq_len_q_b, seq_len_k_b), dtype=int)
mask_ex2[0, 1, 2] = 1 # Batch 0, Query 1, Key 2 is masked
mask_ex2[1, 0, 0] = 1 # Batch 1, Query 0, Key 0 is masked

print("\nQ_ex2 (shape {}):\n".format(Q_ex2.shape), Q_ex2[0,0,:2], "...") # Print a snippet
print("K_ex2 (shape {}):\n".format(K_ex2.shape), K_ex2[0,0,:2], "...")
print("V_ex2 (shape {}):\n".format(V_ex2.shape), V_ex2[0,0,:2], "...")
print("Mask_ex2 (shape {}):\n".format(mask_ex2.shape), mask_ex2)

# Calculate attention
output_ex2, attention_weights_ex2 = scaled_dot_product_attention(Q_ex2, K_ex2, V_ex2, mask=mask_ex

print("\n--- Example 2 Output ---")
print("Attention Weights (shape: {}):\n".format(attention_weights_ex2.shape), attention_weights_ex
print("Output (shape: {}):\n".format(output_ex2.shape), output_ex2)

# Check masked positions in attention weights (should be close to 0)
print("\nChecking masked positions in attention weights:")
print("Batch 0, Query 1, Key 2 weight:", attention_weights_ex2[0, 1, 2])
print("Batch 1, Query 0, Key 0 weight:", attention_weights_ex2[1, 0, 0])

# Expected shapes:
```

```
# Attention Weights: (2, 2, 3)
# Output: (2, 2, 5)
```

Running the Examples:

If you save the Python code (the `softmax` and `scaled_dot_product_attention` functions, and the example calls) into a `.py` file and run it, you will see the computed attention weights and outputs. The attention weights for the masked positions in Example 2 should be very close to zero.

2.5 Key Takeaways

- Scaled Dot-Product Attention computes a weighted sum of `Values`, where the weights are determined by the compatibility of `Queries` and `Keys`.
- Scaling by `sqrt(d_k)` is important for stabilizing gradients.
- Masking allows the model to ignore certain positions during attention calculation, which is crucial for tasks like handling padding or autoregressive decoding.
- Our NumPy implementation can handle both single instances and batches of data.

2.6 What's Next?

With Scaled Dot-Product Attention implemented, we are ready to build a more powerful version: **Multi-Head Attention**. This involves running the scaled dot-product attention mechanism multiple times in parallel with different, learned linear projections of Q, K, and V. We'll cover this in Part 3.

Stay tuned!

Part 3: Building Multi-Head Attention from Scratch

Welcome to Part 3 of our Transformer tutorial! In the previous part, we implemented Scaled

Dot-Product Attention. Now, we'll build upon that to create **Multi-Head Attention**, a key component that enhances the Transformer's ability to focus on different parts of the input sequence in different ways.

3.1 Why Multi-Head Attention?

Single Scaled Dot-Product Attention allows the model to focus on different parts of the sequence based on the (Query, Key, Value) interactions. However, it might be beneficial for the model to attend to different types of information or different "representation subspaces" simultaneously.

Multi-Head Attention achieves this by running the Scaled Dot-Product Attention mechanism multiple times in parallel, each with different, learned linear projections of the Queries, Keys, and Values. This allows each "head" to learn to focus on different aspects of the input.

Benefits:

1. **Diverse Representations:** It allows the model to jointly attend to information from different representation subspaces at different positions. A single attention head might average away some important information, but multiple heads can capture more nuanced relationships.
2. **Stabilized Learning:** Averaging or concatenating the outputs of multiple heads can lead to a more stable and robust attention mechanism.
3. **Increased Model Capacity:** With multiple sets of projection weights, the model has more parameters and potentially a greater capacity to learn complex patterns.

3.2 Mathematical Formulation

Multi-Head Attention consists of several attention heads. For each head `i`, the input Queries (Q), Keys (K), and Values (V) are linearly projected using learned weight matrices:

- `Q_i = Q @ W_i^Q`
- `K_i = K @ W_i^K`
- `V_i = V @ W_i^V`

Where `W_i^Q`, `W_i^K`, and `W_i^V` are the weight matrices for the `i`-th head.

Then, Scaled Dot-Product Attention is applied to each projected set of (Q_i, K_i, V_i) to get the output for that head:

`head_i = Attention(Q_i, K_i, V_i)`

The outputs of all the heads are then concatenated and passed through a final linear projection layer with weight matrix `W^O`:

`MultiHead(Q, K, V) = Concat(head_1, head_2, ..., head_h) @ W^O`

Dimensions:

- Let `d_model` be the input dimension of Q, K, V (e.g., the embedding dimension).
- Let `h` be the number of attention heads.
- The dimensions of the projected queries, keys, and values for each head are typically `d_k = d_model / h` and `d_v = d_model / h`. This ensures that the total computation is similar to a single attention head with full `d_model` dimensions.
    - `W_i^Q` has shape `(d_model, d_k)`
    - `W_i^K` has shape `(d_model, d_k)`
    - `W_i^V` has shape `(d_model, d_v)`
- After concatenation, `Concat(head_1, ..., head_h)` will have shape `(batch_size, seq_len_q, h * d_v)`.
- The output projection matrix `W^O` has shape `(h * d_v, d_model)`, so the final output of Multi-Head Attention has shape `(batch_size, seq_len_q, d_model)`, matching the input query dimension.

3.3 NumPy Implementation

We'll implement Multi-Head Attention as a class to manage the weight matrices and the forward pass.

First, let's re-include the `softmax` and `scaled_dot_product_attention` functions from Part 2, as our `MultiHeadAttention` class will depend on them.

```python
import numpy as np

def softmax(x, axis=-1):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Calculate the attention weights and the output of the attention mechanism.
    Args:
        Q (np.ndarray): Query matrix, shape (..., seq_len_q, d_k)
        K (np.ndarray): Key matrix, shape (..., seq_len_k, d_k)
        V (np.ndarray): Value matrix, shape (..., seq_len_v, d_v) where seq_len_k == seq_len_v
        mask (np.ndarray, optional): Mask to apply to the attention scores. Defaults to None.
                                     Shape should be broadcastable to (..., seq_len_q, seq_len_k).
                                     0 for attend, 1 for mask.
    Returns:
        tuple: (output, attention_weights)
            - output (np.ndarray): The output of the attention mechanism, shape (..., seq_len_q, d
            - attention_weights (np.ndarray): The attention weights, shape (..., seq_len_q, seq_le
    """
    d_k = Q.shape[-1]
    scores = np.matmul(Q, K.swapaxes(-2, -1)) / np.sqrt(d_k)
    if mask is not None:
        # Ensure mask is correctly broadcast. Mask shape (..., seq_len_q, seq_len_k)
        # Add a new axis for num_heads if Q, K, V are (batch, num_heads, seq, dim_k) and mask is (
        # This is typically handled before calling scaled_dot_product_attention if mask is head-ag
        # Or mask can be (batch, num_heads, seq_q, seq_k)
        scores = np.where(mask == 0, scores, -1e9)
    attention_weights = softmax(scores, axis=-1)
    output = np.matmul(attention_weights, V)
    return output, attention_weights

class MultiHeadAttention:
    def __init__(self, d_model, num_heads, seed=None):
        """
        Initialize the MultiHeadAttention layer.
        Args:
            d_model (int): Total dimension of the model.
            num_heads (int): Number of attention heads.
            seed (int, optional): Random seed for weight initialization for reproducibility.
        """
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads # Dimension of keys/queries per head
        self.d_v = d_model // num_heads # Dimension of values per head

        if seed is not None:
            np.random.seed(seed)

        # Weight matrices for input projections (Q, K, V) for all heads combined initially
        # These will be reshaped or split to represent per-head weights.
        # W_q, W_k, W_v transform input (batch, seq_len, d_model) to (batch, seq_len, d_model)
        # which is then reshaped to (batch, seq_len, num_heads, d_k/d_v)
        # and transposed to (batch, num_heads, seq_len, d_k/d_v)
        self.W_q = np.random.randn(d_model, d_model) # Projects input Q to Q' for all heads
        self.W_k = np.random.randn(d_model, d_model) # Projects input K to K' for all heads
        self.W_v = np.random.randn(d_model, d_model) # Projects input V to V' for all heads

        # Weight matrix for output projection
        self.W_o = np.random.randn(d_model, d_model) # Projects concatenated head outputs back to

    def split_heads(self, x, batch_size):
        """
        Split the last dimension into (num_heads, d_k or d_v).
        Then reshape to (batch_size, num_heads, seq_len, d_k or d_v).

        Args:
            x (np.ndarray): Input tensor, shape (batch_size, seq_len, d_model)
            batch_size (int): Batch size.

        Returns:
            np.ndarray: Reshaped tensor, shape (batch_size, num_heads, seq_len, d_k or d_v)
        """
        seq_len = x.shape[1]
        # Reshape to (batch_size, seq_len, num_heads, depth_per_head)
        x = x.reshape(batch_size, seq_len, self.num_heads, -1) # -1 infers d_k or d_v
        # Transpose to (batch_size, num_heads, seq_len, depth_per_head)
        return x.transpose(0, 2, 1, 3)

    def forward(self, Q_in, K_in, V_in, mask=None):
        """
        Perform the forward pass for Multi-Head Attention.

        Args:
            Q_in (np.ndarray): Query tensor, shape (batch_size, seq_len_q, d_model)
            K_in (np.ndarray): Key tensor, shape (batch_size, seq_len_k, d_model)
            V_in (np.ndarray): Value tensor, shape (batch_size, seq_len_v, d_model) (seq_len_k ==
            mask (np.ndarray, optional): Mask to apply. Shape (batch_size, seq_len_q, seq_len_k).
                                         It will be expanded for heads.

        Returns:
            tuple: (output, attention_weights)
                - output (np.ndarray): Final output, shape (batch_size, seq_len_q, d_model)
                - attention_weights (np.ndarray): Attention weights from scaled dot-product attent
                                                  shape (batch_size, num_heads, seq_len_q, seq_len
        """
        batch_size = Q_in.shape[0]
        seq_len_q = Q_in.shape[1]
        # seq_len_k = K_in.shape[1] # (and V)

        # 1. Linear Projections
        # Q_in: (batch_size, seq_len_q, d_model) @ W_q: (d_model, d_model) -> (batch_size, seq_len
        Q_proj = np.matmul(Q_in, self.W_q)
        K_proj = np.matmul(K_in, self.W_k)
        V_proj = np.matmul(V_in, self.W_v)

        # 2. Split heads
        # Resulting shape: (batch_size, num_heads, seq_len_q, d_k)
        Q_split = self.split_heads(Q_proj, batch_size)
        # Resulting shape: (batch_size, num_heads, seq_len_k, d_k)
        K_split = self.split_heads(K_proj, batch_size)
        # Resulting shape: (batch_size, num_heads, seq_len_v, d_v)
        V_split = self.split_heads(V_proj, batch_size)
        # 3. Scaled Dot-Product Attention for each head
        # Q_split: (batch_size, num_heads, seq_len_q, d_k)
        # K_split: (batch_size, num_heads, seq_len_k, d_k)
        # V_split: (batch_size, num_heads, seq_len_v, d_v)
        # Mask needs to be (batch_size, 1, seq_len_q, seq_len_k) to broadcast across heads
        # or (batch_size, num_heads, seq_len_q, seq_len_k) if head-specific mask
        if mask is not None:
            # Expand mask for broadcasting over heads: (batch_size, seq_len_q, seq_len_k) -> (batc
            # This assumes the same mask is applied to all heads.
            mask_expanded = np.expand_dims(mask, axis=1) # Add head dimension
        else:
            mask_expanded = None

        # attention_output shape: (batch_size, num_heads, seq_len_q, d_v)
        # attention_weights shape: (batch_size, num_heads, seq_len_q, seq_len_k)
        attention_output, attention_weights = scaled_dot_product_attention(Q_split, K_split, V_spl

        # 4. Concatenate heads and apply final linear projection
        # Transpose attention_output to (batch_size, seq_len_q, num_heads, d_v)
        attention_output_transposed = attention_output.transpose(0, 2, 1, 3)
        # Reshape to (batch_size, seq_len_q, d_model) because d_model = num_heads * d_v
        concat_attention = attention_output_transposed.reshape(batch_size, seq_len_q, self.d_model

        # Final linear projection: (batch_size, seq_len_q, d_model) @ W_o: (d_model, d_model) -> (
        output = np.matmul(concat_attention, self.W_o)

        return output, attention_weights
```

Explanation of the Code:

1. **`__init__(self, d_model, num_heads, seed=None)`:**
   - Initializes dimensions `d_model`, `num_heads`, `d_k` (depth of key/query per head), and `d_v` (depth of value per head).
   - `d_model` must be divisible by `num_heads`.
   - Initializes weight matrices `W_q`, `W_k`, `W_v` for projecting Q, K, V. Instead of creating `num_heads` separate small matrices, we create larger matrices of shape `(d_model, d_model)`. The output of these projections will then be reshaped and split among heads.

    - Initializes `W_o` of shape `(d_model, d_model)` for the final linear transformation.
    - Weights are initialized with random values from a standard normal distribution (`np.random.randn`). In a real scenario, more sophisticated initialization (like Xavier/Glorot) would be used.
2. **`split_heads(self, x, batch_size)`:**
    - Takes an input `x` of shape `(batch_size, seq_len, d_model)`. This `x` is the result of `Q_in @ W_q` (or K, V equivalents).
    - Reshapes `x` to `(batch_size, seq_len, self.num_heads, self.d_k or self.d_v)`. The `-1` in `reshape` infers the last dimension (which will be `d_k` or `d_v`).
    - Transposes the dimensions to `(batch_size, self.num_heads, seq_len, self.d_k or self.d_v)`. This groups the data by head, making it suitable for parallel attention calculations across heads.
3. **`forward(self, Q_in, K_in, V_in, mask=None)`:**
    - `batch_size = Q_in.shape[0]` and `seq_len_q = Q_in.shape[1]` are extracted.
    - **Linear Projections:** `Q_in`, `K_in`, `V_in` (each of shape `(batch_size, seq_len, d_model)`) are multiplied by their respective weight matrices (`W_q`, `W_k`, `W_v`), resulting in `Q_proj`, `K_proj`, `V_proj` (each of shape `(batch_size, seq_len, d_model)`).
    - **Split Heads:** `Q_proj`, `K_proj`, `V_proj` are passed to `split_heads` to get `Q_split`, `K_split`, `V_split`. Their shapes become `(batch_size, num_heads, seq_len, d_k or d_v)`.
    - **Mask Expansion:** If a `mask` (shape `(batch_size, seq_len_q, seq_len_k)`) is provided, it's expanded to `(batch_size, 1, seq_len_q, seq_len_k)` by adding a new dimension for `num_heads`. This allows the same mask to be broadcast across all heads during the `scaled_dot_product_attention` call.
    - **Scaled Dot-Product Attention:** `scaled_dot_product_attention` is called with `Q_split`, `K_split`, `V_split`, and the `mask_expanded`. This function will perform attention in parallel for all heads because the head dimension is part of the batch dimensions (`...`) that `scaled_dot_product_attention` handles.
        - `attention_output` will have shape `(batch_size, num_heads, seq_len_q, d_v)`.
        - `attention_weights` will have shape `(batch_size, num_heads, seq_len_q, seq_len_k)`.
    - **Concatenate and Project:**
        - `attention_output` is transposed from `(batch_size, num_heads, seq_len_q, d_v)` to `(batch_size, seq_len_q, num_heads, d_v)`. This brings the head dimension

next to the `d_v` dimension.
    - It's then reshaped into `concat_attention` of shape `(batch_size, seq_len_q, self.d_model)`, effectively concatenating the outputs of the heads along the last dimension (since `d_model = num_heads * d_v`).
    - Finally, `concat_attention` is multiplied by `self.W_o` to produce the final `output` of shape `(batch_size, seq_len_q, d_model)`.
- The method returns the final `output` and the `attention_weights` (which can be useful for analysis, showing what each head attended to).

3.4 Simple Input/Output Examples

Let's test our `MultiHeadAttention` class.

```python
# Example Usage of MultiHeadAttention

# Parameters
batch_size_ex = 2
seq_len_q_ex = 4  # Sequence length for queries
seq_len_k_ex = 5  # Sequence length for keys/values (can be different from seq_len_q)
d_model_ex = 12   # Model dimension (e.g., embedding size)
num_heads_ex = 3  # Number of attention heads

# Ensure d_model is divisible by num_heads
assert d_model_ex % num_heads_ex == 0

# Initialize MultiHeadAttention layer
mha = MultiHeadAttention(d_model=d_model_ex, num_heads=num_heads_ex, seed=42)

# Dummy input data
np.random.seed(101)
Q_input = np.random.rand(batch_size_ex, seq_len_q_ex, d_model_ex)
K_input = np.random.rand(batch_size_ex, seq_len_k_ex, d_model_ex)
V_input = np.random.rand(batch_size_ex, seq_len_k_ex, d_model_ex)

# Dummy mask (optional)
# Mask: 0 = attend, 1 = do not attend
# For example, mask out the last key for all queries in the first batch item
# and the first key for all queries in the second batch item.
mask_input = np.zeros((batch_size_ex, seq_len_q_ex, seq_len_k_ex), dtype=int)
mask_input[0, :, -1] = 1 # First batch item, all queries, last key masked
mask_input[1, :, 0] = 1  # Second batch item, all queries, first key masked

print("--- Multi-Head Attention Example ---")
print(f"Input Q shape: {Q_input.shape}")
print(f"Input K shape: {K_input.shape}")
print(f"Input V shape: {V_input.shape}")
print(f"Input Mask shape: {mask_input.shape}")

# Forward pass
output_mha, attention_weights_mha = mha.forward(Q_input, K_input, V_input, mask=mask_input)

print(f"\nOutput MHA shape: {output_mha.shape}")
print(f"Attention Weights MHA shape: {attention_weights_mha.shape}")

# Verify output shape: (batch_size, seq_len_q, d_model)
assert output_mha.shape == (batch_size_ex, seq_len_q_ex, d_model_ex)

# Verify attention_weights shape: (batch_size, num_heads, seq_len_q, seq_len_k)
assert attention_weights_mha.shape == (batch_size_ex, num_heads_ex, seq_len_q_ex, seq_len_k_ex)

print("\nSample of Output MHA (first batch, first query, first 2 features):\n", output_mha[0, 0, :
print("\nSample of Attention Weights (first batch, first head, first query, all keys):\n", attenti

# Check if masking worked (weights for masked positions should be ~0)
print("\nChecking masked attention weights:")
# First batch item, all queries, last key was masked.
# Check weights for first head, first query, last key for the first batch item.
print(f"Weight for masked K (batch 0, head 0, query 0, key {seq_len_k_ex-1}): {attention_weights_m
# Second batch item, all queries, first key was masked.
# Check weights for first head, first query, first key for the second batch item.
print(f"Weight for masked K (batch 1, head 0, query 0, key 0): {attention_weights_mha[1, 0, 0, 0]}

# Example without mask
output_mha_no_mask, _ = mha.forward(Q_input, K_input, V_input, mask=None)
print(f"\nOutput MHA (no mask) shape: {output_mha_no_mask.shape}")
```

Running the Example:

Save the code (including `softmax`, `scaled_dot_product_attention`, the `MultiHeadAttention` class, and the example usage) into a Python file and run it. You should see the shapes of the inputs and outputs, and samples of the results. The attention weights for the masked positions should be very close to zero.

3.5 Key Takeaways

- Multi-Head Attention applies several Scaled Dot-Product Attention operations in parallel.
- Each head uses different linear projections of Q, K, and V, allowing it to learn different aspects of the relationships in the data.
- The outputs of the heads are concatenated and linearly projected to produce the final result.

- This mechanism allows the model to capture a richer set of features and dependencies compared to a single attention head.
- The overall dimensionality `d_model` is typically preserved throughout the Multi-Head Attention block.

3.6 What's Next?

Now that we have Multi-Head Attention, the next crucial component in a Transformer block is the **Position-wise Feed-Forward Network (FFN)**. This is a relatively simple network applied independently to each position in the sequence. We will implement this in Part 4.

Stay tuned!

Part 4: Creating the Position-wise Feed-Forward Layer

Welcome to Part 4 of our NumPy Transformer series! After implementing Scaled Dot-Product Attention (Part 2) and Multi-Head Attention (Part 3), we now turn to another essential component of the Transformer block: the **Position-wise Feed-Forward Network (FFN)**.

4.1 What is a Position-wise Feed-Forward Network?

The Position-wise Feed-Forward Network is a relatively simple component found in each block of the Transformer's encoder and decoder. Its role is to introduce non-linearity and further process the output of the attention sub-layer.

Key characteristics:

1. **Position-wise:** This means the FFN is applied to each position (e.g., each token or word representation in a sequence) independently and identically. The same network (with the same learned weights) is used for every position, but it doesn't share information across

different positions within the same FFN application.
2. **Fully Connected:** It consists of two linear transformations with a non-linear activation function in between.
3. **Fixed Structure:** The structure is consistent across all positions and all Transformer blocks (though the weights are learned separately for each block).

4.2 Mathematical Formulation

The FFN takes an input `x` (which is typically the output of the preceding Multi-Head Attention sub-layer, after residual connection and layer normalization) and transforms it as follows:

`FFN(x) = max(0, x @ W_1 + b_1) @ W_2 + b_2`

Let's break this down:

1. **First Linear Transformation:**
    - `x @ W_1 + b_1`
    - `x` is the input to the FFN for a specific position, with dimension `d_model` (the model's hidden size).
    - `W_1` is the weight matrix of the first linear layer, with shape `(d_model, d_ff)`.
    - `b_1` is the bias vector of the first linear layer, with shape `(d_ff,)`.
    - `d_ff` is the inner-layer dimensionality, often referred to as the feed-forward dimension. Typically, `d_ff` is larger than `d_model` (e.g., `d_ff = 4 * d_model` as in the original Transformer paper).
    - The output of this layer has dimension `d_ff`.
2. **Non-linear Activation (ReLU):**
    - `max(0, ...)`
    - A Rectified Linear Unit (ReLU) activation function is applied element-wise to the output of the first linear layer.
    - `ReLU(z) = max(0, z)`
    - This introduces non-linearity, allowing the model to learn more complex functions.
3. **Second Linear Transformation:**
    - `... @ W_2 + b_2`
    - The output of the ReLU activation (dimension `d_ff`) is then passed through a second linear layer.
    - `W_2` is the weight matrix of the second linear layer, with shape `(d_ff, d_model)`.

- `b_2` is the bias vector of the second linear layer, with shape `(d_model,)`.
- The output of this layer, and thus the output of the FFN, has dimension `d_model`. This brings the representation back to the model's main hidden size, allowing it to be used by subsequent layers or blocks.

When processing a sequence of tokens (e.g., shape `(batch_size, seq_len, d_model)`), this entire FFN operation is applied to each `d_model`-dimensional vector at every position in the sequence independently.

4.3 NumPy Implementation

We'll implement the Position-wise Feed-Forward Network as a class.

```python
import numpy as np

class PositionwiseFeedForward:
    def __init__(self, d_model, d_ff, seed=None):
        """
        Initialize the Position-wise Feed-Forward Network.

        Args:
            d_model (int): Dimension of the input and output.
            d_ff (int): Dimension of the inner layer (feed-forward dimension).
            seed (int, optional): Random seed for weight initialization.
        """
        self.d_model = d_model
        self.d_ff = d_ff

        if seed is not None:
            np.random.seed(seed)

        # Weight matrix for the first linear transformation (d_model -> d_ff)
        # Glorot/Xavier initialization: scale by sqrt(6 / (fan_in + fan_out))
        # For simplicity, using randn here. In practice, better initialization is key.
        limit_w1 = np.sqrt(6.0 / (d_model + d_ff))
        self.W_1 = np.random.uniform(-limit_w1, limit_w1, (d_model, d_ff))
        self.b_1 = np.zeros(d_ff)

        # Weight matrix for the second linear transformation (d_ff -> d_model)
        limit_w2 = np.sqrt(6.0 / (d_ff + d_model))
        self.W_2 = np.random.uniform(-limit_w2, limit_w2, (d_ff, d_model))
        self.b_2 = np.zeros(d_model)

    def relu(self, x):
        """Rectified Linear Unit activation function."""
        return np.maximum(0, x)

    def forward(self, x):
        """
        Perform the forward pass for the Position-wise Feed-Forward Network.

        Args:
            x (np.ndarray): Input tensor, shape (batch_size, seq_len, d_model).

        Returns:
            np.ndarray: Output tensor, shape (batch_size, seq_len, d_model).
        """
        # Input x: (batch_size, seq_len, d_model)

        # First linear transformation
        # x @ W_1: (batch_size, seq_len, d_model) @ (d_model, d_ff) -> (batch_size, seq_len, d_ff)
        # b_1 is broadcasted: (d_ff,) -> (1, 1, d_ff)
        hidden_output = np.matmul(x, self.W_1) + self.b_1

        # ReLU activation
        activated_output = self.relu(hidden_output)

        # Second linear transformation
        # activated_output @ W_2: (batch_size, seq_len, d_ff) @ (d_ff, d_model) -> (batch_size, se
        # b_2 is broadcasted: (d_model,) -> (1, 1, d_model)
        output = np.matmul(activated_output, self.W_2) + self.b_2

        return output
```

Explanation of the Code:

1. **`__init__(self, d_model, d_ff, seed=None)`:**
   - Stores `d_model` (input/output dimension) and `d_ff` (inner feed-forward dimension).
   - Initializes weight matrices `W_1` (shape `(d_model, d_ff)`) and `W_2` (shape `(d_ff, d_model)`), and bias vectors `b_1` (shape `(d_ff,)`) and `b_2` (shape `(d_model,)`).
   - For weight initialization, I've used a simple uniform distribution scaled by a factor derived from Glorot/Xavier initialization (`np.sqrt(6.0 / (fan_in + fan_out))`). Biases are initialized to zeros. In a full deep learning framework, these initializations are often handled automatically with more options.
2. **`relu(self, x)`:**
   - A simple implementation of the ReLU activation function using `np.maximum(0, x)`.
3. **`forward(self, x)`:**
   - Takes an input `x` of shape `(batch_size, seq_len, d_model)`.

- **First Linear Layer:** Computes `np.matmul(x, self.W_1) + self.b_1`. NumPy's broadcasting handles adding `b_1` (shape `(d_ff,)`) to the result of the matrix multiplication (shape `(batch_size, seq_len, d_ff)`).
- **ReLU Activation:** Applies the `relu` function to the output of the first layer.
- **Second Linear Layer:** Computes `np.matmul(activated_output, self.W_2) + self.b_2`. Again, `b_2` (shape `(d_model,)`) is broadcast correctly.
- Returns the final `output` of shape `(batch_size, seq_len, d_model)`.

4.4 Simple Input/Output Examples

Let's test our `PositionwiseFeedForward` class.

```
# Example Usage of PositionwiseFeedForward

# Parameters
batch_size_ex = 2
seq_len_ex = 3     # Sequence length
d_model_ex = 8     # Model dimension (input/output of FFN)
d_ff_ex = 32       # Inner feed-forward dimension (e.g., 4 * d_model)

# Initialize PositionwiseFeedForward layer
pffn = PositionwiseFeedForward(d_model=d_model_ex, d_ff=d_ff_ex, seed=77)

# Dummy input data (e.g., output from a Multi-Head Attention layer)
np.random.seed(102)
input_tensor = np.random.rand(batch_size_ex, seq_len_ex, d_model_ex)

print("--- Position-wise Feed-Forward Network Example ---")
print(f"Input tensor shape: {input_tensor.shape}")

# Forward pass
output_pffn = pffn.forward(input_tensor)

print(f"\nOutput PFFN shape: {output_pffn.shape}")

# Verify output shape: (batch_size, seq_len, d_model)
assert output_pffn.shape == (batch_size_ex, seq_len_ex, d_model_ex)

print("\nSample of Input Tensor (first batch, first token, first 4 features):\n", input_tensor[0,
print("\nSample of Output PFFN (first batch, first token, first 4 features):\n", output_pffn[0, 0,

# Check if the computation is position-wise
# If we process two different positions from the same batch item separately,
# using only their respective d_model vectors, the result for that part should be the same.

# Process position 0 of batch 0
input_pos0_batch0 = input_tensor[0:1, 0:1, :] # Shape (1, 1, d_model)
output_pos0_batch0_isolated = pffn.forward(input_pos0_batch0)

print("\nVerifying position-wise nature:")
print("Output for [0,0] from full batch:", output_pffn[0,0,:4])
print("Output for [0,0] processed isolatedly:", output_pos0_batch0_isolated[0,0,:4])

# They should be very close (potential minor floating point differences if any, but should be iden
assert np.allclose(output_pffn[0,0,:], output_pos0_batch0_isolated[0,0,:])
print("Position-wise check successful.")
```

Running the Example:

If you save the Python code (the `PositionwiseFeedForward` class and the example usage) into a `.py` file and run it, you will see the shapes of the input and output tensors. The example also includes a small check to conceptually verify the "position-wise" nature of the computation by processing a single position vector and comparing it to the corresponding part of the full batch output.

4.5 Key Takeaways

- The Position-wise Feed-Forward Network consists of two linear transformations with a ReLU activation in between.
- It is applied independently to each position in the input sequence.
- It introduces non-linearity and allows for further processing of features after the attention mechanism.
- The input and output dimensions are typically `d_model`, while the inner dimension `d_ff` is larger.

4.6 What's Next?

So far, we have implemented:

1. Scaled Dot-Product Attention
2. Multi-Head Attention
3. Position-wise Feed-Forward Network

Before we can assemble these into a full Transformer Encoder block, we need one more crucial piece: **Positional Encoding**. Since Transformers don't inherently process sequences in order (unlike RNNs), we need to explicitly provide information about the position of each token in the sequence. This will be the focus of Part 5.

Stay tuned!

Part 6: Constructing a Transformer Encoder Block

Welcome to Part 6! We've diligently built all the necessary components:

1. **Scaled Dot-Product Attention** (Part 2)
2. **Multi-Head Attention** (Part 3)
3. **Position-wise Feed-Forward Network** (Part 4)
4. **Positional Encoding** (Part 5)

Now, it's time to assemble these into a complete **Transformer Encoder Block**. The encoder part of the original Transformer is a stack of these identical blocks.

6.1 Architecture of an Encoder Block

A single Transformer Encoder Block consists of two main sub-layers:

1. **Multi-Head Self-Attention Mechanism:** This allows the model to weigh the importance of different words in the input sequence when encoding a particular word.
2. **Position-wise Fully Connected Feed-Forward Network:** This processes the output of the attention mechanism at each position independently.

Around each of these two sub-layers, a **residual connection** is employed, followed by **layer normalization**. This is crucial for training deep networks by preventing vanishing/exploding gradients and stabilizing the learning process.

So, the operations within an encoder block for an input `x` are:

1. **Sub-layer 1 (Multi-Head Attention):**
   - `attention_output = MultiHeadAttention(Q=x, K=x, V=x)` (Self-attention, so Q, K, and

V are all derived from the same input `x`)
    - `x = LayerNorm(x + attention_output)` (Add & Norm)
2. **Sub-layer 2 (Feed-Forward Network):**
    - `ffn_output = PositionwiseFeedForward(x)`
    - `x = LayerNorm(x + ffn_output)` (Add & Norm)

The output of these operations becomes the input to the next encoder block (if any) or the final output of the encoder stack.

Dropout: The original Transformer paper also applies dropout to the output of each sub-layer *before* it is added to the sub-layer input (the residual connection) and normalized. For simplicity in our NumPy-only implementation, we will omit dropout for now, but it's an important regularization technique in practice.

6.2 Layer Normalization

Before we build the encoder block, let's quickly implement Layer Normalization. Unlike Batch Normalization, Layer Normalization normalizes the inputs across the features (i.e., along the `d_model` dimension) for each data sample (each token in each sequence) independently.

Formula:

For a vector `x` (e.g., the `d_model`-dimensional representation of a token):

`LayerNorm(x) = γ * ( (x - μ) / sqrt(σ^2 + ε) ) + β`

Where:

- `μ` is the mean of the elements in `x`.
- `σ^2` is the variance of the elements in `x`.
- `γ` (gamma) is a learnable scaling parameter (vector of size `d_model`).
- `β` (beta) is a learnable shifting parameter (vector of size `d_model`).
- `ε` (epsilon) is a small constant added for numerical stability (e.g., 1e-5).

In our NumPy implementation, we'll initialize `γ` to ones and `β` to zeros, effectively making them non-operational initially. In a full training setup, these would be learned.

```python
import numpy as np

class LayerNormalization:
    def __init__(self, d_model, epsilon=1e-5):
        """
        Initialize the Layer Normalization layer.

        Args:
            d_model (int): Dimension of the model (features dimension).
            epsilon (float): Small constant for numerical stability.
        """
        self.d_model = d_model
        self.epsilon = epsilon
        # Learnable parameters (initialized as if they don't change the input)
        self.gamma = np.ones(d_model)  # Scale
        self.beta = np.zeros(d_model)   # Shift

    def forward(self, x):
        """
        Apply Layer Normalization.

        Args:
            x (np.ndarray): Input tensor, shape (batch_size, seq_len, d_model).

        Returns:
            np.ndarray: Normalized tensor, shape (batch_size, seq_len, d_model).
        """
        # Calculate mean and variance along the last dimension (d_model)
        # keepdims=True is important for broadcasting later
        mean = np.mean(x, axis=-1, keepdims=True)
        variance = np.var(x, axis=-1, keepdims=True)

        # Normalize
        x_normalized = (x - mean) / np.sqrt(variance + self.epsilon)

        # Scale and shift
        # self.gamma and self.beta are (d_model,)
        # They will be broadcasted to (batch_size, seq_len, d_model)
        output = self.gamma * x_normalized + self.beta
        return output
```

6.3 NumPy Implementation of the Encoder Block

Now, let's combine Multi-Head Attention, Position-wise Feed-Forward Network, and Layer Normalization to create the `EncoderBlock` class. We'll need to import or define the classes from previous parts.

For brevity, I'll assume the necessary classes (`MultiHeadAttention`, `PositionwiseFeedForward`) are defined in the same scope or imported. Let's paste their definitions here for completeness within this part's context, along with the `scaled_dot_product_attention` and `softmax` helpers.

```python
# --- Prerequisite code from previous parts --- #
def softmax(x, axis=-1):
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def scaled_dot_product_attention(Q, K, V, mask=None):
    d_k = Q.shape[-1]
    scores = np.matmul(Q, K.swapaxes(-2, -1)) / np.sqrt(d_k)
    if mask is not None:
        scores = np.where(mask == 0, scores, -1e9)
    attention_weights = softmax(scores, axis=-1)
    output = np.matmul(attention_weights, V)
    return output, attention_weights

class MultiHeadAttention:
    def __init__(self, d_model, num_heads, seed=None):
        assert d_model % num_heads == 0
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads
        if seed is not None: np.random.seed(seed)
        self.W_q = np.random.randn(d_model, d_model)
        self.W_k = np.random.randn(d_model, d_model)
        self.W_v = np.random.randn(d_model, d_model)
        self.W_o = np.random.randn(d_model, d_model)

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

    def forward(self, Q_in, K_in, V_in, mask=None):
        batch_size, seq_len_q = Q_in.shape[0], Q_in.shape[1]
        Q_proj = np.matmul(Q_in, self.W_q)
        K_proj = np.matmul(K_in, self.W_k)
        V_proj = np.matmul(V_in, self.W_v)
        Q_split = self.split_heads(Q_proj, batch_size)
        K_split = self.split_heads(K_proj, batch_size)
        V_split = self.split_heads(V_proj, batch_size)
        if mask is not None:
            mask_expanded = np.expand_dims(mask, axis=1)
        else:
            mask_expanded = None
        attention_output, attention_weights = scaled_dot_product_attention(Q_split, K_split, V_spl
        attention_output_transposed = attention_output.transpose(0, 2, 1, 3)
        concat_attention = attention_output_transposed.reshape(batch_size, seq_len_q, self.d_model
        output = np.matmul(concat_attention, self.W_o)
        return output, attention_weights

class PositionwiseFeedForward:
    def __init__(self, d_model, d_ff, seed=None):
        self.d_model = d_model
        self.d_ff = d_ff
        if seed is not None: np.random.seed(seed)
        limit_w1 = np.sqrt(6.0 / (d_model + d_ff))
        self.W_1 = np.random.uniform(-limit_w1, limit_w1, (d_model, d_ff))
        self.b_1 = np.zeros(d_ff)
        limit_w2 = np.sqrt(6.0 / (d_ff + d_model))
        self.W_2 = np.random.uniform(-limit_w2, limit_w2, (d_ff, d_model))
        self.b_2 = np.zeros(d_model)

    def relu(self, x): return np.maximum(0, x)

    def forward(self, x):
        hidden_output = np.matmul(x, self.W_1) + self.b_1
        activated_output = self.relu(hidden_output)
        output = np.matmul(activated_output, self.W_2) + self.b_2
        return output
# --- End of prerequisite code --- #

class EncoderBlock:
    def __init__(self, d_model, num_heads, d_ff, seed=None):
        """
        Initialize a Transformer Encoder Block.

        Args:
            d_model (int): Dimension of the model.
            num_heads (int): Number of attention heads.
            d_ff (int): Dimension of the inner layer in PositionwiseFeedForward.
            seed (int, optional): Random seed for reproducibility of weights.
        """
        if seed is not None: # Manage seed for sub-components
            # Simple way to get different seeds for sub-components from a master seed
            mha_seed = seed
            pffn_seed = seed + 1 if seed is not None else None
            # LayerNorm doesn't have random init in this simple version, but could if gamma/beta w
        else:
            mha_seed = None
            pffn_seed = None

        self.mha = MultiHeadAttention(d_model, num_heads, seed=mha_seed)
        self.ffn = PositionwiseFeedForward(d_model, d_ff, seed=pffn_seed)

        self.layernorm1 = LayerNormalization(d_model)
        self.layernorm2 = LayerNormalization(d_model)

        # Dropout would be initialized here too, e.g., self.dropout_rate = dropout_rate

    def forward(self, x, mask=None):
        """
        Perform the forward pass for the Encoder Block.

        Args:
            x (np.ndarray): Input to the encoder block, shape (batch_size, seq_len, d_model).
            mask (np.ndarray, optional): Mask for the self-attention mechanism.
                                         Shape (batch_size, seq_len_q, seq_len_k).
                                         For self-attention, seq_len_q = seq_len_k = seq_len.

        Returns:
            np.ndarray: Output of the encoder block, shape (batch_size, seq_len, d_model).
        """
        # 1. Multi-Head Self-Attention sub-layer
        # Q=x, K=x, V=x for self-attention
        # The mask here is typically a padding mask for the input sequence.
        attention_output, _ = self.mha.forward(Q_in=x, K_in=x, V_in=x, mask=mask)
        # (Dropout would be applied to attention_output here)
        # Add & Norm
        x_after_mha = self.layernorm1.forward(x + attention_output)
        # 2. Position-wise Feed-Forward sub-layer
        ffn_output = self.ffn.forward(x_after_mha)
        # (Dropout would be applied to ffn_output here)
        # Add & Norm
        output = self.layernorm2.forward(x_after_mha + ffn_output)

        return output
```

Explanation of the Code:

1. **`EncoderBlock.__init__(...)`:**
   - Initializes an instance of `MultiHeadAttention` (`self.mha`).
   - Initializes an instance of `PositionwiseFeedForward` (`self.ffn`).
   - Initializes two instances of `LayerNormalization` (`self.layernorm1` and `self.layernorm2`), one for after the attention sub-layer and one for after the FFN sub-layer.
   - A simple seed management is added to allow reproducible weight initialization in sub-components if a master seed is provided.
2. **`EncoderBlock.forward(self, x, mask=None)`:**
   - Takes the input `x` (shape `(batch_size, seq_len, d_model)`) and an optional `mask`.
   - **Multi-Head Attention Sub-layer:**
     - `attention_output, _ = self.mha.forward(Q_in=x, K_in=x, V_in=x, mask=mask)`: Performs self-attention. The input `x` is used for Queries, Keys, and Values. The `mask` (if provided) is passed to the MHA layer. We only need the output, not the attention weights, for the block's forward pass.
     - `x_after_mha = self.layernorm1.forward(x + attention_output)`: Implements the residual connection (`x + attention_output`) followed by layer normalization.
   - **Position-wise Feed-Forward Sub-layer:**
     - `ffn_output = self.ffn.forward(x_after_mha)`: Passes the output of the first sub-layer through the FFN.
     - `output = self.layernorm2.forward(x_after_mha + ffn_output)`: Implements the second residual connection (`x_after_mha + ffn_output`) followed by layer normalization.
   - Returns the final `output` of the encoder block, which has the same shape as the input `x`.

6.4 Simple Input/Output Examples

Let's test our `EncoderBlock`.

```
# Example Usage of EncoderBlock

# Parameters
batch_size_eb = 2
seq_len_eb = 5      # Input sequence length
d_model_eb = 16     # Model dimension (must be divisible by num_heads)
num_heads_eb = 4    # Number of attention heads
d_ff_eb = 32        # Feed-forward inner dimension (e.g., 2 * d_model or 4 * d_model)

# Initialize EncoderBlock
# Using a seed for reproducible weight initialization in MHA and PFFN
encoder_block = EncoderBlock(d_model=d_model_eb,
                             num_heads=num_heads_eb,
                             d_ff=d_ff_eb,
                             seed=123)

# Dummy input data (e.g., token embeddings + positional encodings)
np.random.seed(456)
input_to_encoder_block = np.random.rand(batch_size_eb, seq_len_eb, d_model_eb)

# Dummy padding mask (optional)
# Mask: 0 = attend, 1 = do not attend
# Example: in the first batch item, the last token is padding.
# In the second batch item, the last two tokens are padding.
# Mask shape should be (batch_size, seq_len_q, seq_len_k)
# For self-attention, seq_len_q = seq_len_k = seq_len_eb
padding_mask = np.zeros((batch_size_eb, seq_len_eb, seq_len_eb), dtype=int)

# Mask for first batch item: last token is padding
# This means the last key is masked for all queries in the first batch item.
padding_mask[0, :, -1] = 1

# Mask for second batch item: last two tokens are padding
padding_mask[1, :, -2:] = 1

# Note: A more typical padding mask might be simpler, e.g., (batch_size, 1, seq_len_k)
# if it only depends on keys. Our MHA expects (batch_size, seq_len_q, seq_len_k)
# or for it to be broadcastable. The current mask structure is fine for self-attention.
```

```
print("--- Transformer Encoder Block Example ---")
print(f"Input tensor shape: {input_to_encoder_block.shape}")
if padding_mask is not None:
    print(f"Padding mask shape: {padding_mask.shape}")

# Forward pass through the encoder block
output_encoder_block = encoder_block.forward(input_to_encoder_block, mask=padding_mask)
# output_encoder_block = encoder_block.forward(input_to_encoder_block, mask=None) # Test without m

print(f"\nOutput of Encoder Block shape: {output_encoder_block.shape}")

# Verify output shape: (batch_size, seq_len, d_model)
assert output_encoder_block.shape == (batch_size_eb, seq_len_eb, d_model_eb)

print("\nSample of Input (first batch, first token, first 4 features):\n", input_to_encoder_block[
print("\nSample of Output (first batch, first token, first 4 features):\n", output_encoder_block[0

# The internal attention weights could also be inspected if mha.forward returned them
# and EncoderBlock also returned them, e.g., for visualization.
```

Running the Example:

Save all the class definitions (`LayerNormalization` , `MultiHeadAttention` , `PositionwiseFeedForward` , `EncoderBlock` , and helpers) and the example usage into a Python file. Running it will demonstrate a forward pass through a single encoder block, showing that the output shape matches the input shape (`d_model` is preserved).

6.5 Key Takeaways

- A Transformer Encoder Block is composed of a Multi-Head Self-Attention sub-layer and a Position-wise Feed-Forward Network sub-layer.
- Residual connections and Layer Normalization are applied around each sub-layer, which are critical for training deep models.
- The output of an encoder block has the same dimensionality (`d_model`) as its input, allowing multiple blocks to be stacked.
- Dropout is typically used for regularization but was omitted here for simplicity.

6.6 What's Next?

With a single `EncoderBlock` implemented, the next step (Part 7) is to create the **Full Transformer Encoder**, which involves stacking multiple `EncoderBlock` instances. We will also discuss how the initial input embeddings and positional encodings are fed into this stack.

Stay tuned!

Part 7: Implementing the Full Transformer Encoder

Welcome to Part 7 of our "Transformer from Scratch with NumPy" series. In Part 6, we constructed a single `EncoderBlock`. Now, we'll take that building block and stack multiple instances of it to create the complete Transformer Encoder. We will also integrate the input embeddings (conceptually) and the positional encodings (from Part 5) that prepare the input sequence for the encoder stack.

7.1 Introduction to the Full Encoder

The encoder side of the Transformer is responsible for processing an input sequence of tokens and transforming it into a sequence of continuous representations (contextual embeddings). These representations ideally capture the meaning and context of each token in relation to the entire input sequence.

The Transformer encoder achieves this by using a stack of `N` identical `EncoderBlock`s. The output of one block becomes the input to the next. This layered approach allows the model to learn increasingly complex features and relationships within the data.

Overall Architecture:

1. **Input Embeddings:** The input sequence of tokens (e.g., words) is first converted into dense vector representations (embeddings).
2. **Positional Encoding:** Since the Transformer architecture itself doesn't inherently process

sequential order (due to the self-attention mechanism operating on sets of vectors), positional information is added to the embeddings.
3. **Encoder Stack:** The sum of embeddings and positional encodings is then fed into the stack of `N` `EncoderBlock`s.

7.2 Input to the Encoder

7.2.1 Input Embeddings

In a typical NLP pipeline, raw text is first tokenized (e.g., split into words or sub-words). Each token is then mapped to a high-dimensional vector using an embedding layer. These embeddings are usually learned during the training process.

For this tutorial series, we are focusing on the Transformer architecture itself and will assume that the input to our encoder is already a sequence of numerical embeddings. For example, if our input sentence has `seq_len` tokens and our model uses an embedding dimension of `d_model`, the input to the encoder (before positional encoding) will be a NumPy array of shape `(batch_size, seq_len, d_model)`.

7.2.2 Positional Encoding

As discussed in Part 5, positional encodings are crucial for providing the model with information about the relative or absolute position of tokens in the sequence. We implemented the `get_positional_encoding` function using sinusoidal functions.

The positional encodings have the same dimension `d_model` as the token embeddings. They are added element-wise to the token embeddings before being fed into the first encoder block.

Let's recall the `get_positional_encoding` function from Part 5:

```python
import numpy as np

def get_positional_encoding(seq_len, d_model):
    """
    Generates sinusoidal positional encodings.
    Args:
        seq_len (int): Length of the sequence.
        d_model (int): Dimension of the model (embedding dimension).
    Returns:
        np.ndarray: Positional encoding matrix of shape (1, seq_len, d_model).
                    The first dimension is 1 to allow broadcasting with batch_size.
    """
    position = np.arange(seq_len)[:, np.newaxis] # (seq_len, 1)
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model)) # (d_model/2)

    pe = np.zeros((seq_len, d_model))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)

    return pe[np.newaxis, :, :] # Shape: (1, seq_len, d_model)
```

7.3 The `TransformerEncoder` Class

We will now define a `TransformerEncoder` class that encapsulates the stack of `N` `EncoderBlock`s. This class will:

1. Initialize `N` instances of `EncoderBlock`.
2. In its `forward` method:
   a. Take the input embeddings.
   b. Add positional encodings to these embeddings.
   c. Pass the result sequentially through each `EncoderBlock` in the stack.

For the `EncoderBlock` itself, we'll use the implementation from Part 6. For simplicity in this file, we'll redefine the `EncoderBlock` and its necessary components (`LayerNormalization`, `dropout_layer`, and dummy versions of `MultiHeadAttention` and `PositionwiseFeedForward`) so the example is self-contained.

7.4 NumPy Implementation of `TransformerEncoder`

First, let's bring in the necessary components. We'll use the `LayerNormalization` and `dropout_layer` from Part 6, and simplified (dummy) versions of `MultiHeadAttention` and `PositionwiseFeedForward` for the `EncoderBlock` example to keep it concise. In a real scenario, you'd import these from their respective part files.

```python
import numpy as np

# --- Positional Encoding (from Part 5) ---
def get_positional_encoding(seq_len, d_model):
    position = np.arange(seq_len)[:, np.newaxis]
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))
    pe = np.zeros((seq_len, d_model))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)
    return pe[np.newaxis, :, :]

# --- LayerNormalization class (from Part 6) ---
class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.d_model = d_model
        self.eps = eps
        self.gamma = np.ones(d_model)
        self.beta = np.zeros(d_model)

    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        variance = np.var(x, axis=-1, keepdims=True)
        x_normalized = (x - mean) / np.sqrt(variance + self.eps)
        output = self.gamma * x_normalized + self.beta
        return output

# --- dropout_layer function (from Part 6) ---
def dropout_layer(x, rate, training=True):
    if not training or rate == 0:
        return x
    mask = np.random.binomial(1, 1 - rate, size=x.shape) / (1 - rate)
    return x * mask

# --- Dummy MultiHeadAttention (Placeholder for Part 3's implementation) ---
class DummyMultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
    def forward(self, query, key, value, mask=None):
        batch_size, seq_len_q, _ = query.shape
        # Simulate MHA output (just returns random data of correct shape)
        output = np.random.rand(batch_size, seq_len_q, self.d_model)
        # Dummy attention weights
        seq_len_k = key.shape[1]
        attention_weights = np.random.rand(batch_size, self.num_heads, seq_len_q, seq_len_k)
        return output, attention_weights

# --- Dummy PositionwiseFeedForward (Placeholder for Part 4's implementation) ---
class DummyPositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model
        self.d_ff = d_ff

    def forward(self, x):
        # Simulate PFF output (just returns random data of correct shape)
        return np.random.rand(x.shape[0], x.shape[1], self.d_model)

# --- EncoderBlock class (from Part 6) ---
class EncoderBlock:
    def __init__(self, multi_head_attention, positionwise_feed_forward, d_model, dropout_rate):
        self.multi_head_attention = multi_head_attention
        self.positionwise_feed_forward = positionwise_feed_forward
        self.norm1 = LayerNormalization(d_model)
        self.norm2 = LayerNormalization(d_model)
        self.dropout_rate = dropout_rate

    def forward(self, x, mask, training=True):
        attn_output, _ = self.multi_head_attention.forward(query=x, key=x, value=x, mask=mask)
        attn_output_dropout = dropout_layer(attn_output, self.dropout_rate, training)
        norm1_input = x + attn_output_dropout
        norm1_output = self.norm1.forward(norm1_input)

        ffn_output = self.positionwise_feed_forward.forward(norm1_output)
        ffn_output_dropout = dropout_layer(ffn_output, self.dropout_rate, training)
        norm2_input = norm1_output + ffn_output_dropout
        output = self.norm2.forward(norm2_input)
        return output
```

```
# --- Full Transformer Encoder ---
class TransformerEncoder:
    def __init__(self, num_blocks, d_model, num_heads, d_ff, dropout_rate, max_seq_len):
        """
        Initializes the Transformer Encoder.
        Args:
            num_blocks (int): Number of EncoderBlocks to stack.
            d_model (int): Dimension of the model (embedding dimension).
            num_heads (int): Number of attention heads for MultiHeadAttention.
            d_ff (int): Dimension of the hidden layer in PositionwiseFeedForward.
            dropout_rate (float): Dropout rate.
            max_seq_len (int): Maximum sequence length for positional encoding.
        """
        self.d_model = d_model
        self.num_blocks = num_blocks
        self.dropout_rate = dropout_rate

        # Positional encoding - precompute for max_seq_len
        self.pos_encoding = get_positional_encoding(max_seq_len, d_model)

        # Create a list of EncoderBlocks
        # In a real implementation, MHA and PFF would be properly initialized with weights.
        # Here, we use the dummy versions for simplicity of the example.
        self.encoder_blocks = []
        for _ in range(num_blocks):
            mha_instance = DummyMultiHeadAttention(d_model, num_heads)
            pff_instance = DummyPositionwiseFeedForward(d_model, d_ff)
            encoder_block = EncoderBlock(mha_instance, pff_instance, d_model, dropout_rate)
            self.encoder_blocks.append(encoder_block)

    def forward(self, x, mask, training=True):
        """
        Forward pass for the Transformer Encoder.
        Args:
            x (np.ndarray): Input tensor (embeddings), shape (batch_size, seq_len, d_model).
            mask (np.ndarray): Mask for the Multi-Head Attention layers in EncoderBlocks.
                               Shape should be compatible (e.g., (batch_size, 1, seq_len, seq_len)
            training (bool): Whether the model is in training mode (for dropout).
        Returns:
            np.ndarray: Output tensor from the encoder, shape (batch_size, seq_len, d_model).
        """
        batch_size, seq_len, _ = x.shape

        # 1. Add positional encoding
        # x is (batch_size, seq_len, d_model)
        # self.pos_encoding is (1, max_seq_len, d_model)
        # We need to slice pos_encoding to match current seq_len and add to x.
        x = x + self.pos_encoding[:, :seq_len, :]

        # Apply dropout to the embeddings + positional encoding
        x = dropout_layer(x, self.dropout_rate, training)

        # 2. Pass through the stack of EncoderBlocks
        for i in range(self.num_blocks):
            x = self.encoder_blocks[i].forward(x, mask, training)

        return x
```

7.5 Simple Input/Output Example

Let's test our `TransformerEncoder` with some dummy data.

```
# --- Example Usage ---
np.random.seed(123) # For reproducibility

# Parameters
batch_size = 2
seq_len = 15        # Length of the input sequence
d_model = 128       # Embedding dimension
num_heads = 8       # Number of attention heads
d_ff = 256          # Dimension of FFN hidden layer
dropout_rate = 0.1
num_encoder_blocks = 3 # Number of EncoderBlocks in the stack
max_seq_len_for_pe = 50 # Max sequence length for precomputed positional encoding

# Instantiate the Transformer Encoder
transformer_encoder = TransformerEncoder(
    num_blocks=num_encoder_blocks,
    d_model=d_model,
    num_heads=num_heads,
    d_ff=d_ff,
    dropout_rate=dropout_rate,
    max_seq_len=max_seq_len_for_pe
)

# Create dummy input tensor (e.g., token embeddings)
# Shape: (batch_size, seq_len, d_model)
input_embeddings = np.random.rand(batch_size, seq_len, d_model)

# Create a dummy mask (e.g., no padding, so all tokens attend to each other)
# Shape: (batch_size, 1, seq_len_q, seq_len_k)
# For self-attention in encoder, seq_len_q = seq_len_k = seq_len
encoder_attention_mask = np.zeros((batch_size, 1, seq_len, seq_len), dtype=int)

# Pass input through the Transformer Encoder
# Set training=False for this example to make dropout a pass-through for consistent output
encoder_output = transformer_encoder.forward(input_embeddings, mask=encoder_attention_mask, traini

print("--- Transformer Encoder Example ---")
print("Input embeddings shape:", input_embeddings.shape)
print("Encoder attention mask shape:", encoder_attention_mask.shape)
print("Positional encoding shape (sliced for current seq_len):".format(seq_len), transformer_encod
print("Output tensor shape:", encoder_output.shape)
print("
Sample of Output tensor (first item in batch, first token, first 5 features):
", encoder_output[0, 0, :5])

# Verify output shape
if encoder_output.shape == (batch_size, seq_len, d_model):
    print("
Output shape is correct!")
else:
    print("
Output shape is INCORRECT!")
```

Running the Example:
If you combine all the Python code blocks (Positional Encoding, LayerNorm, Dropout, Dummy MHA, Dummy PFF, EncoderBlock, TransformerEncoder, and the example usage) into a single script and run it, you will see the output. The `encoder_output` will have the same shape as the input embeddings `(batch_size, seq_len, d_model)`, which is `(2, 15, 128)` in this example. This demonstrates that the entire encoder stack processes the sequence and produces representations of the same dimensionality.

7.6 Key Takeaways

- The Transformer Encoder is composed of a stack of `N` identical `EncoderBlock`s.
- Input to the encoder first involves converting tokens to embeddings and then adding positional encodings.
- The `TransformerEncoder` class manages this stack and the initial addition of positional encodings.
- Each `EncoderBlock` within the stack applies self-attention and feed-forward operations, refining the representations at each step.

7.7 What's Next?

With the encoder fully implemented, we are ready to move to the other major component of the original Transformer architecture: the Decoder. In **Part 8: Implementing the Transformer Decoder Block**, we will build the `DecoderBlock`, which has a slightly different structure than the `EncoderBlock` due to its need to attend to the encoder's output and handle masked self-attention for auto-regressive generation. Stay tuned!

Part 8: Implementing the Transformer Decoder Block

Welcome to Part 8 of our "Transformer from Scratch with NumPy" series! Having built the `TransformerEncoder` in Part 7, we now turn our attention to its counterpart: the Transformer Decoder. In this part, we will construct a single `DecoderBlock`. The decoder has a slightly more complex structure than the encoder block because it needs to handle two types of attention mechanisms and generate an output sequence token by token.

8.1 Introduction to the Decoder Block

The decoder's role in the Transformer is to generate an output sequence (e.g., a translated sentence) based on the encoded representation of the input sequence. It does this auto-regressively, meaning it generates one token at a time, and the previously generated tokens are used as input to generate the next token.

A Transformer Decoder is typically composed of a stack of `N` identical `DecoderBlock`s. Each `DecoderBlock` has three main sub-layers:

1. **Masked Multi-Head Self-Attention Mechanism:** This allows the decoder to attend to different positions in the output sequence generated so far. The "masked" part is crucial: it ensures that when predicting a token at position `i`, the self-attention mechanism can only attend to tokens at positions less than `i`. This prevents the model from "cheating" by looking at future tokens it hasn't predicted yet.
2. **Encoder-Decoder Multi-Head Attention Mechanism:** This layer allows the decoder to attend to the output of the encoder (the contextualized representations of the input sequence). For each token being generated by the decoder, this mechanism helps decide

which parts of the input sequence are most relevant.

3. **Position-wise Feed-Forward Network (FFN):** Similar to the encoder, this is a fully connected feed-forward network applied independently to each position.

Like in the encoder block, residual connections are employed around each of these three sub-layers, followed by layer normalization.

The overall structure for one decoder block can be visualized as:

```
Target Input -> Masked Multi-Head Attention -> Add & Norm -> Encoder-Decoder Attention -> Add & Nor
```

8.2 Components of the Decoder Block

We will reuse several components we've already built:

- `MultiHeadAttention` (from Part 3): This will be used for both self-attention and encoder-decoder attention.
- `PositionwiseFeedForward` (from Part 4).
- `LayerNormalization` (from Part 6).
- `dropout_layer` (from Part 6).

8.2.1 Masked Multi-Head Self-Attention

This is a standard multi-head self-attention mechanism, but with a crucial difference: a **look-ahead mask** (or subsequent mask) is applied. This mask prevents positions from attending to subsequent positions. For a target sequence of length `seq_len_tgt`, the mask will be an upper triangular matrix that, when applied, zeros out (or sets to a large negative number before softmax) the attention scores for future tokens.

8.2.2 Encoder-Decoder Attention

In this sub-layer:

- **Queries (Q)** come from the output of the previous sub-layer in the decoder (the masked multi-head self-attention layer).
- **Keys (K)** and **Values (V)** come from the output of the `TransformerEncoder` (i.e., the encoded representation of the source sequence).

This allows every position in the decoder to attend over all positions in the input sequence. The mask used here is typically a **padding mask** if the input sequence has padding, to prevent attention to `<pad>` tokens in the source.

8.3 NumPy Implementation of `DecoderBlock`

Let's define the `DecoderBlock` class. We'll bring in the necessary components again for a self-contained example. In a full project, you would import these.

```python
import numpy as np

# --- LayerNormalization class (from Part 6) ---
class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.d_model = d_model
        self.eps = eps
        self.gamma = np.ones(d_model)
        self.beta = np.zeros(d_model)

    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        variance = np.var(x, axis=-1, keepdims=True)
        x_normalized = (x - mean) / np.sqrt(variance + self.eps)
        output = self.gamma * x_normalized + self.beta
        return output

# --- dropout_layer function (from Part 6) ---
def dropout_layer(x, rate, training=True):
    if not training or rate == 0:
        return x
    mask = np.random.binomial(1, 1 - rate, size=x.shape) / (1 - rate)
    return x * mask

# --- Dummy MultiHeadAttention (Placeholder for Part 3's implementation) ---
# In a real implementation, this would be the actual MHA from Part 3.
class DummyMultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        # W_q, W_k, W_v, W_o would be initialized here.

    def forward(self, query, key, value, mask=None):
        # query: (batch_size, seq_len_q, d_model)
        # key:   (batch_size, seq_len_k, d_model)
        # value: (batch_size, seq_len_v, d_model) where seq_len_k == seq_len_v
        batch_size, seq_len_q, _ = query.shape
        # Simulate MHA output
        output = np.random.rand(batch_size, seq_len_q, self.d_model)
        # Dummy attention weights (batch_size, num_heads, seq_len_q, seq_len_k)
        seq_len_k = key.shape[1]
        attention_weights = np.random.rand(batch_size, self.num_heads, seq_len_q, seq_len_k)
        return output, attention_weights

# --- Dummy PositionwiseFeedForward (Placeholder for Part 4's implementation) ---
class DummyPositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model
        self.d_ff = d_ff
        # Weights and biases for two linear layers would be here.

    def forward(self, x):
        # x: (batch_size, seq_len, d_model)
        # Simulate PFF output
        return np.random.rand(x.shape[0], x.shape[1], self.d_model)

# --- DecoderBlock class ---
class DecoderBlock:
    def __init__(self, self_attention_mha, encoder_decoder_attention_mha,
                 positionwise_feed_forward, d_model, dropout_rate):
        """
        Initializes a Transformer Decoder Block.
        Args:
            self_attention_mha: MultiHeadAttention instance for masked self-attention.
            encoder_decoder_attention_mha: MultiHeadAttention instance for encoder-decoder attenti
            positionwise_feed_forward: PositionwiseFeedForward instance.
            d_model (int): The dimension of the model (embedding dimension).
            dropout_rate (float): The dropout rate.
        """
        self.self_attention_mha = self_attention_mha
        self.encoder_decoder_attention_mha = encoder_decoder_attention_mha
        self.positionwise_feed_forward = positionwise_feed_forward

        self.norm1 = LayerNormalization(d_model)
        self.norm2 = LayerNormalization(d_model)
        self.norm3 = LayerNormalization(d_model)

        self.dropout_rate = dropout_rate
    def forward(self, target_x, encoder_output, look_ahead_mask, padding_mask, training=True):
        """
        Forward pass for the Decoder Block.
        Args:
            target_x (np.ndarray): Input tensor for the decoder (e.g., target embeddings + pos_enc
                                   shape (batch_size, target_seq_len, d_model).
            encoder_output (np.ndarray): Output from the TransformerEncoder,
                                         shape (batch_size, source_seq_len, d_model).
            look_ahead_mask (np.ndarray): Mask for the self-attention sub-layer to prevent attendi
                                          Shape (batch_size, 1, target_seq_len, target_seq_len).
            padding_mask (np.ndarray): Mask for the encoder-decoder attention sub-layer to hide pa
                                       Shape (batch_size, 1, target_seq_len, source_seq_len).
            training (bool): Whether the model is in training mode (for dropout).
        Returns:
            np.ndarray: Output tensor, shape (batch_size, target_seq_len, d_model).
            np.ndarray: Attention weights from the self-attention mechanism.
            np.ndarray: Attention weights from the encoder-decoder attention mechanism.
        """
        # 1. Masked Multi-Head Self-Attention sub-layer
        # Q, K, V are all derived from target_x
        self_attn_output, self_attn_weights = self.self_attention_mha.forward(
            query=target_x, key=target_x, value=target_x, mask=look_ahead_mask
        )
        self_attn_output_dropout = dropout_layer(self_attn_output, self.dropout_rate, training)
        # Add residual connection and LayerNorm
        norm1_input = target_x + self_attn_output_dropout
        norm1_output = self.norm1.forward(norm1_input)

        # 2. Encoder-Decoder Multi-Head Attention sub-layer
        # Q comes from norm1_output (output of previous sub-layer)
        # K, V come from encoder_output
        enc_dec_attn_output, enc_dec_attn_weights = self.encoder_decoder_attention_mha.forward(
            query=norm1_output, key=encoder_output, value=encoder_output, mask=padding_mask
        )
        enc_dec_attn_output_dropout = dropout_layer(enc_dec_attn_output, self.dropout_rate, traini
        # Add residual connection and LayerNorm
        norm2_input = norm1_output + enc_dec_attn_output_dropout
        norm2_output = self.norm2.forward(norm2_input)

        # 3. Position-wise Feed-Forward sub-layer
        ffn_output = self.positionwise_feed_forward.forward(norm2_output)
        ffn_output_dropout = dropout_layer(ffn_output, self.dropout_rate, training)
        # Add residual connection and LayerNorm
        norm3_input = norm2_output + ffn_output_dropout
        output = self.norm3.forward(norm3_input)

        return output, self_attn_weights, enc_dec_attn_weights
```

8.4 Creating Masks for the Decoder

Two types of masks are typically needed for the decoder block:

1. **Look-Ahead Mask (Subsequent Mask):** This is used in the self-attention mechanism of the decoder. It prevents any position from attending to subsequent positions. This is essential for auto-regressive generation, as the prediction for the current token should not depend on future tokens.
   A look-ahead mask for a sequence of length `L` is an `L x L` matrix where the upper triangle (elements `(i, j)` where `j > i`) is masked (e.g., set to 1 if 1 means mask, or a large negative number if applied before softmax).
2. **Padding Mask:** This mask is used in the encoder-decoder attention mechanism. If the source sequence (input to the encoder) was padded to a fixed length, this mask ensures that the decoder does not attend to these `<pad>` tokens in the encoder's output.
   It also applies to the target sequence in the masked self-attention if the target sequence itself has padding.

Let's define a helper function to create a look-ahead mask.

```
def create_look_ahead_mask(seq_len):
    """
    Creates a look-ahead mask for self-attention in the decoder.
    The mask is an upper triangular matrix of 1s (or True) where attention should be prevented.
    Args:
        seq_len (int): Length of the target sequence.
    Returns:
        np.ndarray: Look-ahead mask of shape (seq_len, seq_len).
                    Mask value 1 (or True) means do not attend.
    """
    mask = 1 - np.triu(np.ones((seq_len, seq_len)), k=1) # Lower triangle and diagonal are 1, uppe
    mask = np.triu(np.ones((seq_len, seq_len)), k=1) # Upper triangle is 1 (mask), rest is 0
    return mask # Shape: (seq_len, seq_len). 1 means mask, 0 means attend.
                # This needs to be broadcastable to (batch_size, num_heads, seq_len, seq_len)
                # Our scaled_dot_product_attention expects mask == 0 to attend, mask != 0 to preve
                # So, a mask value of 1 will correctly be interpreted as "prevent attention".

# Example of look_ahead_mask for seq_len = 3:
# [[0, 1, 1],
#  [0, 0, 1],
#  [0, 0, 0]]
# This means:
# - Token 0 can attend to token 0.
# - Token 1 can attend to tokens 0, 1.
# - Token 2 can attend to tokens 0, 1, 2.
```

A padding mask is simpler: it would typically be generated based on where the `<pad>` tokens are in the input sequences. For example, if `input_sequence` has `<pad>` tokens represented by 0, then `padding_mask = (input_sequence == 0)`. This boolean mask then needs to be reshaped and broadcast appropriately for the attention mechanism (e.g., `(batch_size, 1, 1, source_seq_len)` for encoder-decoder attention).

8.5 Simple Input/Output Example

Let's test our `DecoderBlock` with dummy data.

```python
# --- Example Usage ---
np.random.seed(4242)

# Parameters
batch_size = 2
target_seq_len = 12 # Length of the target sequence
source_seq_len = 10 # Length of the source sequence (from encoder)
d_model = 64       # Embedding dimension
num_heads = 4      # Number of attention heads
d_ff = 128         # Dimension of FFN hidden layer
dropout_rate = 0.1

# Instantiate dummy components
self_mha = DummyMultiHeadAttention(d_model, num_heads)
enc_dec_mha = DummyMultiHeadAttention(d_model, num_heads)
pff_instance = DummyPositionwiseFeedForward(d_model, d_ff)

# Instantiate the Decoder Block
decoder_block = DecoderBlock(self_mha, enc_dec_mha, pff_instance, d_model, dropout_rate)

# Create dummy input tensors
# Target input (e.g., shifted target embeddings + positional encoding)
# Shape: (batch_size, target_seq_len, d_model)
target_input_tensor = np.random.rand(batch_size, target_seq_len, d_model)

# Encoder output (from the TransformerEncoder)
# Shape: (batch_size, source_seq_len, d_model)
encoder_output_tensor = np.random.rand(batch_size, source_seq_len, d_model)

# Create masks
# 1. Look-ahead mask for self-attention
# Shape: (target_seq_len, target_seq_len)
look_ahead_mask_single = create_look_ahead_mask(target_seq_len)
# Reshape for batch and head broadcasting: (batch_size, 1, target_seq_len, target_seq_len)
# The MultiHeadAttention implementation should handle broadcasting from (target_seq_len, target_se
# or expect this full shape. Our dummy MHA doesn't use the mask values, but a real one would.
# For scaled_dot_product_attention, mask is (..., seq_len_q, seq_len_k)
# So, (batch_size, 1, target_seq_len, target_seq_len) is a common shape.
look_ahead_mask_batch = np.tile(look_ahead_mask_single[np.newaxis, np.newaxis, :, :], (batch_size,
# 2. Padding mask for encoder-decoder attention (e.g., no padding in encoder output for this examp
# This mask prevents attention to padded tokens in the *source* sequence (encoder_output).
# Shape: (batch_size, 1, target_seq_len, source_seq_len)
# If no padding, mask is all zeros (attend to everything).
padding_mask_batch = np.zeros((batch_size, 1, target_seq_len, source_seq_len), dtype=int)

# Pass inputs through the decoder block
# Set training=False for consistent output (dropout is pass-through)
decoder_output, self_attn_w, enc_dec_attn_w = decoder_block.forward(
    target_input_tensor,
    encoder_output_tensor,
    look_ahead_mask_batch,
    padding_mask_batch,
    training=False
)

print("--- Decoder Block Example ---")
print("Target input tensor shape:", target_input_tensor.shape)
print("Encoder output tensor shape:", encoder_output_tensor.shape)
print("Look-ahead mask (batch) shape:", look_ahead_mask_batch.shape)
print("Padding mask (batch) shape:", padding_mask_batch.shape)
print("Decoder output tensor shape:", decoder_output.shape)
print("Self-attention weights shape:", self_attn_w.shape) # (batch_size, num_heads, target_seq_len
print("Encoder-decoder attention weights shape:", enc_dec_attn_w.shape) # (batch_size, num_heads,

print("
Sample of Decoder output (first item, first token, first 5 features):
", decoder_output[0, 0, :5])

# Verify output shape
if decoder_output.shape == (batch_size, target_seq_len, d_model):
    print("
Decoder output shape is correct!")
else:
    print("
Decoder output shape is INCORRECT!")

# Check a value in the look-ahead mask
print(f"
Look-ahead mask for target_seq_len={target_seq_len} (sample):")
print(look_ahead_mask_single)
# For scaled_dot_product_attention, if mask[b, h, q_idx, k_idx] != 0, then attention is prevented.
# So, look_ahead_mask_single[0,1] should be 1 (prevent Q0 from attending to K1).
# And look_ahead_mask_single[1,0] should be 0 (allow Q1 to attend to K0).
```

Running the Example:
When you run the combined Python code, you'll observe the shapes of all inputs, masks, and outputs. The `decoder_output` will have the shape `(batch_size, target_seq_len, d_model)`, which is `(2, 12, 64)` in this example. The attention weight shapes will also be printed, corresponding to the self-attention and encoder-decoder attention mechanisms.

8.6 Key Takeaways

- A `DecoderBlock` has three main sub-layers: Masked Multi-Head Self-Attention, Encoder-Decoder Multi-Head Attention, and a Position-wise Feed-Forward Network.
- Residual connections and Layer Normalization are applied around each sub-layer.
- The **masked self-attention** uses a look-ahead mask to ensure auto-regressive behavior (preventing attention to future tokens).
- The **encoder-decoder attention** allows the decoder to focus on relevant parts of the encoded input sequence, using the encoder's output as Keys and Values.
- Appropriate masking (look-ahead and padding) is crucial for the correct functioning of the decoder.

8.7 What's Next?

With both the `EncoderBlock` (from Part 6/7) and the `DecoderBlock` now defined, we are ready to assemble the full Transformer model. In **Part 9: Building the Complete Transformer Architecture**, we will combine the `TransformerEncoder` (stack of `EncoderBlock`s), a `TransformerDecoder` (stack of `DecoderBlock`s), input/output embeddings, and a final linear layer to create the end-to-end Transformer model. Stay tuned!

Part 9: Building the Complete Transformer Architecture

Welcome to Part 9 of our "Transformer from Scratch with NumPy" series! We've meticulously constructed all the core components: attention mechanisms, feed-forward networks, positional encodings, encoder blocks, and decoder blocks. Now, it's time to assemble them into the full end-to-end Transformer model as described in the original "Attention Is All You Need" paper.

9.1 Overview of the Full Transformer Architecture

The Transformer model consists of two main parts: an **Encoder** and a **Decoder**.

- **Encoder:** Processes the input sequence (e.g., a sentence in the source language) and generates a sequence of continuous representations (contextual embeddings). It's made up of a stack of identical Encoder Blocks.
- **Decoder:** Takes the encoder's output and the target sequence (e.g., the sentence in the target language, shifted right during training) to generate the output sequence token by token in an auto-regressive manner. It's made up of a stack of identical Decoder Blocks.

The overall data flow is as follows:

1. **Input Embeddings:** Source and target sequences are converted into dense vector embeddings.
2. **Positional Encoding:** Positional information is added to these embeddings.
3. **Encoder Pass:** The source embeddings (with positional encoding) are passed through the encoder stack.
4. **Decoder Pass:** The target embeddings (with positional encoding and appropriate masking) and the encoder's output are passed through the decoder stack.
5. **Final Linear Layer & Softmax:** The decoder's output is passed through a final linear layer to project it to the size of the target vocabulary, followed by a softmax function to obtain probability distributions over the vocabulary for each output token.

9.2 Core Components Recap

We will be using:

- **Embedding Layer:** To convert input token IDs into dense vectors. (We'll define a simple one).
- **Positional Encoding:** `get_positional_encoding` (from Part 5).
- **TransformerEncoder:** (from Part 7) - a stack of `EncoderBlock`s.
- **TransformerDecoder:** (We'll define this) - a stack of `DecoderBlock`s.
- **EncoderBlock:** (from Part 6) - contains Multi-Head Attention and Position-wise FFN.
- **DecoderBlock:** (from Part 8) - contains Masked Multi-Head Self-Attention, Encoder-Decoder Attention, and Position-wise FFN.
- **MultiHeadAttention:** (from Part 3).
- **PositionwiseFeedForward:** (from Part 4).
- **LayerNormalization:** (from Part 6).
- **Final Output Layer:** A linear transformation followed by softmax.

For the NumPy implementation within this markdown, we'll use dummy/simplified versions of `MultiHeadAttention` and `PositionwiseFeedForward` within the `EncoderBlock` and `DecoderBlock` definitions to keep the example focused and concise. The text will assume you'd use the full versions from previous parts in a complete implementation.

9.3 NumPy Implementation

Let's start by bringing in and defining the necessary components.

```
import numpy as np

# --- Softmax (from Part 2) ---
def softmax(x, axis=-1):
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

# --- Positional Encoding (from Part 5) ---
def get_positional_encoding(seq_len, d_model):
    position = np.arange(seq_len)[:, np.newaxis]
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))
    pe = np.zeros((seq_len, d_model))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)
    return pe[np.newaxis, :, :]

# --- LayerNormalization (from Part 6) ---
class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.d_model = d_model
        self.eps = eps
        self.gamma = np.ones(d_model) # Learnable scale
        self.beta = np.zeros(d_model)  # Learnable shift

    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        variance = np.var(x, axis=-1, keepdims=True)
        x_normalized = (x - mean) / np.sqrt(variance + self.eps)
        return self.gamma * x_normalized + self.beta

# --- Dropout Layer (from Part 6) ---
def dropout_layer(x, rate, training=True):
    if not training or rate == 0:
        return x
    mask = np.random.binomial(1, 1 - rate, size=x.shape) / (1 - rate)
    return x * mask

# --- Dummy MultiHeadAttention & PositionwiseFeedForward (for brevity in this example) ---
class DummyMultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        # In a real MHA, W_q, W_k, W_v, W_o matrices (learnable) would be here.
        # For simplicity, forward will return random data of correct shape.

    def forward(self, query, key, value, mask=None):
        batch_size, seq_len_q, _ = query.shape
        output = np.random.rand(batch_size, seq_len_q, self.d_model) # (batch, seq_len_q, d_model)
        seq_len_k = key.shape[1]
        attention_weights = np.random.rand(batch_size, self.num_heads, seq_len_q, seq_len_k)
        return output, attention_weights

class DummyPositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model
        self.d_ff = d_ff
        # Real PFF would have two linear layers with weights and biases (learnable).

    def forward(self, x):
        return np.random.rand(x.shape[0], x.shape[1], self.d_model)

# --- EncoderBlock (from Part 6, using Dummies) ---
class EncoderBlock:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate):
        self.mha = DummyMultiHeadAttention(d_model, num_heads)
        self.pff = DummyPositionwiseFeedForward(d_model, d_ff)
        self.norm1 = LayerNormalization(d_model)
        self.norm2 = LayerNormalization(d_model)
        self.dropout_rate = dropout_rate

    def forward(self, x, mask, training=True):
        attn_output, _ = self.mha.forward(x, x, x, mask)
        attn_output = dropout_layer(attn_output, self.dropout_rate, training)
        out1 = self.norm1.forward(x + attn_output)

        pff_output = self.pff.forward(out1)
        pff_output = dropout_layer(pff_output, self.dropout_rate, training)
        out2 = self.norm2.forward(out1 + pff_output)
        return out2
```

```
# --- TransformerEncoder (from Part 7) ---
class TransformerEncoder:
    def __init__(self, num_blocks, d_model, num_heads, d_ff, dropout_rate):
        self.d_model = d_model
        self.num_blocks = num_blocks
        self.encoder_blocks = [EncoderBlock(d_model, num_heads, d_ff, dropout_rate) for _ in range
        self.dropout_rate = dropout_rate # For dropout on embeddings+PE

    def forward(self, x_emb, mask, training=True):
        # x_emb is already (batch_size, seq_len, d_model) including positional encoding
        x = dropout_layer(x_emb, self.dropout_rate, training)
        for i in range(self.num_blocks):
            x = self.encoder_blocks[i].forward(x, mask, training)
        return x

# --- DecoderBlock (from Part 8, using Dummies) ---
class DecoderBlock:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate):
        self.masked_mha = DummyMultiHeadAttention(d_model, num_heads)
        self.encoder_decoder_mha = DummyMultiHeadAttention(d_model, num_heads)
        self.pff = DummyPositionwiseFeedForward(d_model, d_ff)
        self.norm1 = LayerNormalization(d_model)
        self.norm2 = LayerNormalization(d_model)
        self.norm3 = LayerNormalization(d_model)
        self.dropout_rate = dropout_rate

    def forward(self, target_x, encoder_output, look_ahead_mask, padding_mask, training=True):
        # Masked Self-Attention
        attn1_out, _ = self.masked_mha.forward(target_x, target_x, target_x, look_ahead_mask)
        attn1_out = dropout_layer(attn1_out, self.dropout_rate, training)
        out1 = self.norm1.forward(target_x + attn1_out)

        # Encoder-Decoder Attention
        attn2_out, _ = self.encoder_decoder_mha.forward(out1, encoder_output, encoder_output, padd
        attn2_out = dropout_layer(attn2_out, self.dropout_rate, training)
        out2 = self.norm2.forward(out1 + attn2_out)

        # Position-wise Feed-Forward
        pff_output = self.pff.forward(out2)
```

```python
        pff_output = dropout_layer(pff_output, self.dropout_rate, training)
        out3 = self.norm3.forward(out2 + pff_output)
        return out3

# --- TransformerDecoder (New: Stacks DecoderBlocks) ---
class TransformerDecoder:
    def __init__(self, num_blocks, d_model, num_heads, d_ff, dropout_rate):
        self.d_model = d_model
        self.num_blocks = num_blocks
        self.decoder_blocks = [DecoderBlock(d_model, num_heads, d_ff, dropout_rate) for _ in range
        self.dropout_rate = dropout_rate # For dropout on embeddings+PE

    def forward(self, target_x_emb, encoder_output, look_ahead_mask, padding_mask, training=True):
        # target_x_emb is already (batch_size, target_seq_len, d_model) including positional encod
        x = dropout_layer(target_x_emb, self.dropout_rate, training)
        for i in range(self.num_blocks):
            x = self.decoder_blocks[i].forward(x, encoder_output, look_ahead_mask, padding_mask, t
        return x

# --- Simple Embedding Layer (New) ---
class EmbeddingLayer:
    def __init__(self, vocab_size, d_model):
        self.vocab_size = vocab_size
        self.d_model = d_model
        # Initialize embedding matrix with small random values
        # This would be learned during actual training.
        self.embeddings = np.random.randn(vocab_size, d_model) * 0.01

    def forward(self, token_ids):
        # token_ids: (batch_size, seq_len)
        # Output: (batch_size, seq_len, d_model)
        return self.embeddings[token_ids]

# --- Final Linear Layer (New) ---
class FinalLinearLayer:
    def __init__(self, d_model, target_vocab_size):
        self.d_model = d_model
        self.target_vocab_size = target_vocab_size
        # Initialize weights and bias (learnable)
        self.weights = np.random.randn(d_model, target_vocab_size) * 0.01
        self.bias = np.zeros(target_vocab_size)

    def forward(self, x):
        # x: (batch_size, target_seq_len, d_model)
        # Output: (batch_size, target_seq_len, target_vocab_size)
        return np.dot(x, self.weights) + self.bias

# --- The Full Transformer Model (New) ---
class Transformer:
    def __init__(self, num_encoder_blocks, num_decoder_blocks,
                 d_model, num_heads, d_ff,
                 source_vocab_size, target_vocab_size,
                 max_seq_len, dropout_rate=0.1):

        self.max_seq_len = max_seq_len
        self.d_model = d_model

        self.source_embedding = EmbeddingLayer(source_vocab_size, d_model)
        self.target_embedding = EmbeddingLayer(target_vocab_size, d_model)

        self.pos_encoding_source = get_positional_encoding(max_seq_len, d_model)
        self.pos_encoding_target = get_positional_encoding(max_seq_len, d_model)

        self.encoder = TransformerEncoder(num_encoder_blocks, d_model, num_heads, d_ff, dropout_ra
        self.decoder = TransformerDecoder(num_decoder_blocks, d_model, num_heads, d_ff, dropout_ra

        self.final_linear_layer = FinalLinearLayer(d_model, target_vocab_size)
        self.dropout_rate = dropout_rate # General dropout for embeddings

    def forward(self, source_tokens, target_tokens, source_padding_mask, target_padding_mask, look
        """
        Forward pass for the full Transformer model.
        Args:
            source_tokens (np.ndarray): Source sequence token IDs, shape (batch_size, source_seq_l
            target_tokens (np.ndarray): Target sequence token IDs, shape (batch_size, target_seq_l
            source_padding_mask (np.ndarray): Mask for padding in source sequence for encoder MHA.
                                              Shape (batch_size, 1, source_seq_len, source_seq_len).
            target_padding_mask (np.ndarray): Mask for padding in target sequence for decoder's en
                                              Shape (batch_size, 1, target_seq_len, source_seq_len).
            look_ahead_mask (np.ndarray): Mask for decoder's self-attention.
```

```
                                          Shape (batch_size, 1, target_seq_len, target_seq_len).
            training (bool): If True, applies dropout.
        Returns:
            np.ndarray: Output logits, shape (batch_size, target_seq_len, target_vocab_size).
        """
        source_seq_len = source_tokens.shape[1]
        target_seq_len = target_tokens.shape[1]

        # 1. Source Embeddings + Positional Encoding
        source_emb = self.source_embedding.forward(source_tokens)
        # Scale embeddings as per original paper (optional but common)
        source_emb *= np.sqrt(self.d_model)
        source_emb_pe = source_emb + self.pos_encoding_source[:, :source_seq_len, :]
        source_emb_pe = dropout_layer(source_emb_pe, self.dropout_rate, training)

        # 2. Target Embeddings + Positional Encoding
        target_emb = self.target_embedding.forward(target_tokens)
        target_emb *= np.sqrt(self.d_model)
        target_emb_pe = target_emb + self.pos_encoding_target[:, :target_seq_len, :]
        target_emb_pe = dropout_layer(target_emb_pe, self.dropout_rate, training)

        # 3. Encoder Pass
        encoder_output = self.encoder.forward(source_emb_pe, source_padding_mask, training)

        # 4. Decoder Pass
        # The padding_mask for decoder's MHA2 should be based on source sequence padding.
        decoder_output = self.decoder.forward(target_emb_pe, encoder_output, look_ahead_mask, targ

        # 5. Final Linear Layer and Softmax (Softmax is usually applied outside, with the loss fun
        output_logits = self.final_linear_layer.forward(decoder_output)

        return output_logits
```

9.4 Mask Creation for the Full Model

When using the Transformer, creating the correct masks is crucial:

1. **Source Padding Mask (`source_padding_mask`):**

- Used in the Encoder's self-attention layers.
- Masks out `<pad>` tokens in the source sequence.
- Shape: `(batch_size, 1, 1, source_seq_len)` or `(batch_size, 1, source_seq_len, source_seq_len)` if it's a combined mask for all queries attending to keys.
- If `token_id == padding_token_id`, then mask value is 1 (prevent attention), else 0.

2. **Target Look-Ahead Mask (`look_ahead_mask`):**
    - Used in the Decoder's first (masked) self-attention layer.
    - Prevents positions from attending to subsequent positions in the target sequence.
    - Combines a standard look-ahead mask (upper triangle) with the target padding mask (if any).
    - Shape: `(batch_size, 1, target_seq_len, target_seq_len)`.
3. **Target Padding Mask (for Encoder-Decoder Attention - `target_padding_mask` in `forward` args):**
    - Used in the Decoder's second (encoder-decoder) attention layer.
    - Masks out `<pad>` tokens from the *source sequence* (i.e., `encoder_output`).
    - Shape: `(batch_size, 1, 1, source_seq_len)` or `(batch_size, 1, target_seq_len, source_seq_len)`.
    - This ensures the decoder doesn't attend to padded parts of the encoder's output.

Helper function to create masks (simplified for example):

```
def create_padding_mask_for_encoder_self_attn(sequence_padded, pad_token_id):
    # sequence_padded: (batch_size, seq_len)
    mask = (sequence_padded == pad_token_id).astype(int)
    return mask[:, np.newaxis, np.newaxis, :] # (batch_size, 1, 1, seq_len)

def create_look_ahead_mask_for_decoder_self_attn(target_seq_len):
    # Creates the upper triangular mask for subsequent positions
    look_ahead = np.triu(np.ones((target_seq_len, target_seq_len)), k=1).astype(int)
    return look_ahead[np.newaxis, np.newaxis, :, :] # (1, 1, target_seq_len, target_seq_len), broa

def create_combined_decoder_mask(target_sequence_padded, pad_token_id):
    # target_sequence_padded: (batch_size, target_seq_len)
    target_seq_len = target_sequence_padded.shape[1]

    # 1. Look-ahead part (prevents seeing future tokens)
    look_ahead = np.triu(np.ones((target_seq_len, target_seq_len)), k=1).astype(int)
    # (target_seq_len, target_seq_len)

    # 2. Padding part for target sequence (prevents seeing padding in target itself)
    target_pad = (target_sequence_padded == pad_token_id).astype(int)[:, np.newaxis, :]
    # (batch_size, 1, target_seq_len)

    # Combine: if either look_ahead is 1 OR target_pad is 1 for the key, then mask.
    # The mask should be (batch_size, 1, target_seq_len, target_seq_len)
    # A position (q, k) is masked if look_ahead[q, k] == 1 OR target_pad[batch, k] == 1
    combined_mask = np.maximum(look_ahead, target_pad)
    return combined_mask[:, np.newaxis, :, :]
```

Note: The `scaled_dot_product_attention` function from Part 2 expects mask values of `0` to attend and non-zero (e.g., `1`) to prevent attention. The mask creation functions above follow this convention.

9.5 Simple Input/Output Example

Let's instantiate and run our full Transformer model.

```
# --- Example Usage ---
np.random.seed(1337)

# Model Hyperparameters
num_enc_blocks = 2
num_dec_blocks = 2
d_model_hyper = 64 # Must be divisible by num_heads
num_heads_hyper = 4
d_ff_hyper = 128
source_vocab_size_hyper = 1000 # Example source vocabulary size
target_vocab_size_hyper = 1200 # Example target vocabulary size
max_seq_len_hyper = 50       # Max sequence length for positional encoding
dropout_rate_hyper = 0.1

# Instantiate the Transformer
transformer_model = Transformer(
    num_encoder_blocks=num_enc_blocks,
    num_decoder_blocks=num_dec_blocks,
    d_model=d_model_hyper,
    num_heads=num_heads_hyper,
    d_ff=d_ff_hyper,
    source_vocab_size=source_vocab_size_hyper,
    target_vocab_size=target_vocab_size_hyper,
    max_seq_len=max_seq_len_hyper,
    dropout_rate=dropout_rate_hyper
)

# Dummy Input Data
batch_size_ex = 2
source_seq_len_ex = 10
target_seq_len_ex = 12 # For training, target usually shifted input
pad_token_id_ex = 0

# Source and Target token sequences (random integers representing token IDs)
# Replace with actual tokenized data in a real scenario.
# Ensure token IDs are < vocab_size
source_tokens_ex = np.random.randint(1, source_vocab_size_hyper, size=(batch_size_ex, source_seq_l
source_tokens_ex[0, -2:] = pad_token_id_ex # Add some padding to the first batch item
```

```
target_tokens_ex = np.random.randint(1, target_vocab_size_hyper, size=(batch_size_ex, target_seq_l
target_tokens_ex[1, -3:] = pad_token_id_ex # Add some padding to the second batch item

# Create Masks (Simplified for this example)
# 1. Source Padding Mask (for Encoder Self-Attention)
#    Masks <pad> tokens in the source. Shape (batch, 1, 1, src_len)
src_padding_mask_ex = (source_tokens_ex == pad_token_id_ex)[:, np.newaxis, np.newaxis, :].astype(i

# 2. Target Look-Ahead Mask (for Decoder Self-Attention)
#    Prevents attending to future tokens and <pad> tokens in the target.
#    Shape (batch, 1, tgt_len, tgt_len)
tgt_look_ahead_mask_triu = np.triu(np.ones((target_seq_len_ex, target_seq_len_ex)), k=1).astype(in
tgt_padding_component = (target_tokens_ex == pad_token_id_ex)[:, np.newaxis, :].astype(int) # (bat
tgt_look_ahead_mask_ex = np.maximum(tgt_look_ahead_mask_triu, tgt_padding_component)[:, np.newaxis

# 3. Encoder Output Padding Mask (for Decoder Encoder-Decoder Attention)
#    Masks <pad> tokens in the source (encoder output) when decoder attends to it.
#    Shape (batch, 1, 1, src_len) - this will broadcast over target_seq_len queries.
#    Or (batch, 1, tgt_len, src_len) if you want to be explicit.
enc_out_padding_mask_ex = (source_tokens_ex == pad_token_id_ex)[:, np.newaxis, np.newaxis, :].asty


print("--- Full Transformer Example ---")
print("Source Tokens (shape {}):
".format(source_tokens_ex.shape), source_tokens_ex)
print("Target Tokens (shape {}):
".format(target_tokens_ex.shape), target_tokens_ex)
print("Source Padding Mask (shape {}):
".format(src_padding_mask_ex.shape), src_padding_mask_ex[0,:,:,:5]) # Print snippet
print("Target Look-Ahead Mask (shape {}):
".format(tgt_look_ahead_mask_ex.shape), tgt_look_ahead_mask_ex[0,:,:3,:3]) # Print snippet
print("Encoder Output Padding Mask (shape {}):
".format(enc_out_padding_mask_ex.shape), enc_out_padding_mask_ex[0,:,:,:5]) # Print snippet

# Forward pass (training=False for consistent dropout behavior in example)
output_logits_ex = transformer_model.forward(
    source_tokens_ex,
    target_tokens_ex,
    src_padding_mask_ex,      # For encoder's self-attention
    enc_out_padding_mask_ex,  # For decoder's enc-dec attention (masking encoder output)
    tgt_look_ahead_mask_ex,   # For decoder's self-attention
    training=False
)

print("
Output Logits shape:", output_logits_ex.shape)
# Expected: (batch_size_ex, target_seq_len_ex, target_vocab_size_hyper)
# (2, 12, 1200)

# Apply softmax to get probabilities (usually done by loss function)
output_probs_ex = softmax(output_logits_ex, axis=-1)
print("Output Probabilities shape:", output_probs_ex.shape)
print("Sample Probabilities (sum over vocab for 1st token, 1st batch item):", np.sum(output_probs_

if output_logits_ex.shape == (batch_size_ex, target_seq_len_ex, target_vocab_size_hyper):
    print("
Output logits shape is correct!")
else:
    print("
Output logits shape is INCORRECT!")
```

Running the Example:
Executing this code will initialize the full Transformer model with dummy weights and pass some sample data through it. You'll see the shapes of inputs, masks, and the final output logits. The output logits will have the shape `(batch_size, target_seq_len, target_vocab_size)`, ready to be used with a cross-entropy loss function during training.

9.6 Key Takeaways

- The full Transformer model elegantly combines an encoder and a decoder stack.
- Input and target sequences are first embedded and augmented with positional encodings.
- The encoder processes the source sequence to produce contextual representations.
- The decoder uses these representations and the (shifted) target sequence to auto-regressively generate output tokens.
- A final linear layer followed by softmax converts decoder outputs into probability

distributions over the target vocabulary.
- Correctly implementing and applying masks (padding and look-ahead) is absolutely critical for the Transformer to function as intended.

9.7 What's Next?

We have now built the entire Transformer architecture from scratch using NumPy! This is a significant achievement. The final step in our series, **Part 10: Putting It All Together: Training on a Toy Dataset**, will discuss the conceptual steps involved in training this model. While a full NumPy-based training loop with backpropagation for all these components is beyond the scope of a single tutorial part (it's a very deep network!), we'll outline the process, define a simple loss function, and discuss how one might approach gradient updates if implementing the backward pass manually. This will provide a conceptual understanding of how such a model learns.

Stay tuned for the grand finale!

Part 10: Putting It All Together: Training on a Toy Dataset and Conclusion

Welcome to the grand finale, Part 10 of our "Transformer from Scratch with NumPy" series! We have successfully built all the individual components and assembled the full Transformer architecture in Part 9. Now, we'll discuss how one would go about training such a model, demonstrate a forward pass on a toy dataset, calculate a loss, and conclude our journey.

Disclaimer: Implementing a full, efficient training loop with backpropagation for a complex model like the Transformer *entirely from scratch in NumPy* is a monumental task, primarily undertaken for deep educational purposes or in resource-constrained environments. Modern deep learning frameworks (PyTorch, TensorFlow) automate gradient calculation and offer optimized operations, making them the practical choice for real-world training. This part will focus on the conceptual understanding.

10.1 The Training Process: An Overview

Training a sequence-to-sequence model like the Transformer typically involves the following steps:

1. **Dataset Preparation:**
    - Collect a large corpus of paired sequences (e.g., English sentences and their French translations).
    - Tokenize the sequences: Convert text into sequences of numerical IDs based on a vocabulary.
    - Create input and target pairs: For a translation task, the source sentence is the input, and the target sentence is what the model should predict.
    - Handle padding: Ensure all sequences in a batch have the same length by adding special `<pad>` tokens.
    - Create masks: Generate padding masks and look-ahead masks.
2. **Model Initialization:**
    - Instantiate the Transformer model with chosen hyperparameters (number of layers, `d_model`, `num_heads`, `d_ff`, vocab sizes, dropout rate).
    - Initialize the model's learnable parameters (weights and biases in embedding layers, linear projections in MHA and PFF, LayerNorm parameters, final linear layer). This was done with small random values in our dummy implementations.
3. **Training Loop:** Iterate for a chosen number of epochs:
   a. **Batching:** Divide the dataset into mini-batches.
   b. For each batch:
   i. **Forward Pass:** Feed the source and target sequences (and their masks) into the model to get output logits.
   ii. **Loss Calculation:** Compare the model's output logits with the actual target sequences to compute a loss. Cross-entropy loss is standard for classification tasks like predicting the next token.
   iii. **Backward Pass (Backpropagation):** Calculate the gradients of the loss with respect to all learnable parameters in the model. This is the most complex part to implement manually.
   iv. **Parameter Update:** Adjust the model's parameters using an optimizer (e.g., Adam, SGD) and the calculated gradients to minimize the loss.
4. **Evaluation:** Periodically evaluate the model on a separate validation set to monitor

performance and prevent overfitting.

5. **Inference/Prediction:** Once trained, use the model to generate output sequences for new, unseen input sequences.

10.2 A Toy Dataset Example: Reversing a Sequence of Numbers

Let's define a very simple task: teaching the Transformer to reverse a sequence of numbers. For example, if the input is `[1, 2, 3]`, the output should be `[3, 2, 1]`.

- **Vocabulary:** `0` (pad), `1` (SOS - start of sequence), `2` (EOS - end of sequence), `3, 4, 5, 6, 7` (our numbers).
  - Source Vocab Size: 8
  - Target Vocab Size: 8
- **Example Pair:**
  - Source: `[1, 3, 4, 5, 2]` (SOS, 1, 2, 3, EOS)
  - Target (for training input to decoder - shifted right): `[1, 5, 4, 3, 2]` (SOS, 3, 2, 1, EOS)
  - Target (for loss calculation): `[5, 4, 3, 2, 2]` (3, 2, 1, EOS, EOS - we predict EOS after the sequence)

```python
import numpy as np

# --- Re-include necessary components from Part 9 for a runnable snippet ---
# Softmax, Positional Encoding, LayerNorm, Dropout, Dummy MHA, Dummy PFF,
# EncoderBlock, TransformerEncoder, DecoderBlock, TransformerDecoder,
# EmbeddingLayer, FinalLinearLayer, Transformer (full model)

def softmax(x, axis=-1):
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def get_positional_encoding(seq_len, d_model):
    position = np.arange(seq_len)[:, np.newaxis]
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))
    pe = np.zeros((seq_len, d_model))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)
    return pe[np.newaxis, :, :]

class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.d_model = d_model; self.eps = eps
        self.gamma = np.ones(d_model); self.beta = np.zeros(d_model)
    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        variance = np.var(x, axis=-1, keepdims=True)
        x_normalized = (x - mean) / np.sqrt(variance + self.eps)
        return self.gamma * x_normalized + self.beta

def dropout_layer(x, rate, training=True):
    if not training or rate == 0: return x
    mask = np.random.binomial(1, 1 - rate, size=x.shape) / (1 - rate)
    return x * mask

class DummyMultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model; self.num_heads = num_heads
        # Dummy weights for illustration (not actually used in this simplified forward)
        self.W_q = [np.random.randn(d_model, d_model // num_heads) for _ in range(num_heads)]
        self.W_k = [np.random.randn(d_model, d_model // num_heads) for _ in range(num_heads)]
        self.W_v = [np.random.randn(d_model, d_model // num_heads) for _ in range(num_heads)]
        self.W_o = np.random.randn(d_model, d_model)

    def forward(self, query, key, value, mask=None):
        batch_size, seq_len_q, _ = query.shape
        output = np.random.rand(batch_size, seq_len_q, self.d_model)
        seq_len_k = key.shape[1]
        attention_weights = np.random.rand(batch_size, self.num_heads, seq_len_q, seq_len_k)
        return output, attention_weights

class DummyPositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model; self.d_ff = d_ff
        # Dummy weights
        self.W1 = np.random.randn(d_model, d_ff)
        self.b1 = np.zeros(d_ff)
        self.W2 = np.random.randn(d_ff, d_model)
        self.b2 = np.zeros(d_model)
    def forward(self, x):
        # Simplified: just return random of correct shape
        return np.random.rand(x.shape[0], x.shape[1], self.d_model)

class EncoderBlock:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate):
        self.mha = DummyMultiHeadAttention(d_model, num_heads)
        self.pff = DummyPositionwiseFeedForward(d_model, d_ff)
        self.norm1 = LayerNormalization(d_model); self.norm2 = LayerNormalization(d_model)
        self.dropout_rate = dropout_rate
    def forward(self, x, mask, training=True):
        attn_output, _ = self.mha.forward(x, x, x, mask)
        attn_output = dropout_layer(attn_output, self.dropout_rate, training)
        out1 = self.norm1.forward(x + attn_output)
        pff_output = self.pff.forward(out1)
        pff_output = dropout_layer(pff_output, self.dropout_rate, training)
        out2 = self.norm2.forward(out1 + pff_output)
        return out2

class TransformerEncoder:
    def __init__(self, num_blocks, d_model, num_heads, d_ff, dropout_rate):
        self.encoder_blocks = [EncoderBlock(d_model, num_heads, d_ff, dropout_rate) for _ in range
        self.dropout_rate = dropout_rate
    def forward(self, x_emb, mask, training=True):
        x = dropout_layer(x_emb, self.dropout_rate, training)
        for block in self.encoder_blocks: x = block.forward(x, mask, training)
        return x

class DecoderBlock:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate):
        self.masked_mha = DummyMultiHeadAttention(d_model, num_heads)
        self.encoder_decoder_mha = DummyMultiHeadAttention(d_model, num_heads)
        self.pff = DummyPositionwiseFeedForward(d_model, d_ff)
        self.norm1 = LayerNormalization(d_model); self.norm2 = LayerNormalization(d_model); self.n
        self.dropout_rate = dropout_rate
    def forward(self, target_x, encoder_output, look_ahead_mask, padding_mask, training=True):
        attn1_out, _ = self.masked_mha.forward(target_x, target_x, target_x, look_ahead_mask)
        attn1_out = dropout_layer(attn1_out, self.dropout_rate, training)
        out1 = self.norm1.forward(target_x + attn1_out)
        attn2_out, _ = self.encoder_decoder_mha.forward(out1, encoder_output, encoder_output, padd
        attn2_out = dropout_layer(attn2_out, self.dropout_rate, training)
        out2 = self.norm2.forward(out1 + attn2_out)
        pff_output = self.pff.forward(out2)
        pff_output = dropout_layer(pff_output, self.dropout_rate, training)
        out3 = self.norm3.forward(out2 + pff_output)
        return out3

class TransformerDecoder:
    def __init__(self, num_blocks, d_model, num_heads, d_ff, dropout_rate):
        self.decoder_blocks = [DecoderBlock(d_model, num_heads, d_ff, dropout_rate) for _ in range
        self.dropout_rate = dropout_rate
    def forward(self, target_x_emb, encoder_output, look_ahead_mask, padding_mask, training=True):
        x = dropout_layer(target_x_emb, self.dropout_rate, training)
        for block in self.decoder_blocks: x = block.forward(x, encoder_output, look_ahead_mask, pa
        return x

class EmbeddingLayer:
    def __init__(self, vocab_size, d_model):
        self.embeddings = np.random.randn(vocab_size, d_model) * 0.01
    def forward(self, token_ids):
        return self.embeddings[token_ids]
```

```
class FinalLinearLayer:
    def __init__(self, d_model, target_vocab_size):
        self.weights = np.random.randn(d_model, target_vocab_size) * 0.01
        self.bias = np.zeros(target_vocab_size)
    def forward(self, x):
        return np.dot(x, self.weights) + self.bias

class Transformer:
    def __init__(self, num_encoder_blocks, num_decoder_blocks, d_model, num_heads, d_ff,
                 source_vocab_size, target_vocab_size, max_seq_len, dropout_rate=0.1):
        self.d_model = d_model; self.max_seq_len = max_seq_len; self.dropout_rate = dropout_rate
        self.source_embedding = EmbeddingLayer(source_vocab_size, d_model)
        self.target_embedding = EmbeddingLayer(target_vocab_size, d_model)
        self.pos_encoding_source = get_positional_encoding(max_seq_len, d_model)
        self.pos_encoding_target = get_positional_encoding(max_seq_len, d_model)
        self.encoder = TransformerEncoder(num_encoder_blocks, d_model, num_heads, d_ff, dropout_ra
        self.decoder = TransformerDecoder(num_decoder_blocks, d_model, num_heads, d_ff, dropout_ra
        self.final_linear_layer = FinalLinearLayer(d_model, target_vocab_size)

    def forward(self, source_tokens, target_tokens, source_padding_mask, target_padding_mask, look
        source_seq_len = source_tokens.shape[1]; target_seq_len = target_tokens.shape[1]
        source_emb = self.source_embedding.forward(source_tokens) * np.sqrt(self.d_model)
        source_emb_pe = dropout_layer(source_emb + self.pos_encoding_source[:, :source_seq_len, :]
        target_emb = self.target_embedding.forward(target_tokens) * np.sqrt(self.d_model)
        target_emb_pe = dropout_layer(target_emb + self.pos_encoding_target[:, :target_seq_len, :]
        encoder_output = self.encoder.forward(source_emb_pe, source_padding_mask, training)
        decoder_output = self.decoder.forward(target_emb_pe, encoder_output, look_ahead_mask, targ
        return self.final_linear_layer.forward(decoder_output)

# --- Toy Dataset & Parameters ---
PAD_ID = 0
SOS_ID = 1 # Start Of Sequence
EOS_ID = 2 # End Of Sequence
NUM_START_ID = 3 # Actual numbers start from ID 3

# Vocabulary: 0:PAD, 1:SOS, 2:EOS, 3:"0", 4:"1", ..., 12:"9"
# For simplicity, let's use numbers 0-6 (IDs 3 to 9)
VOCAB_SIZE = NUM_START_ID + 7 # 3 + 7 = 10 (0-6 are 7 numbers)
```

```
# Hyperparameters for the toy model
d_model_toy = 32
num_heads_toy = 2
d_ff_toy = 64
num_enc_blocks_toy = 1
num_dec_blocks_toy = 1
max_seq_len_toy = 10 # Max length of sequence like [SOS, n1, n2, n3, EOS]
dropout_toy = 0.0 # No dropout for this simple example

# Instantiate the Transformer for the toy task
toy_transformer = Transformer(
    num_encoder_blocks=num_enc_blocks_toy,
    num_decoder_blocks=num_dec_blocks_toy,
    d_model=d_model_toy,
    num_heads=num_heads_toy,
    d_ff=d_ff_toy,
    source_vocab_size=VOCAB_SIZE,
    target_vocab_size=VOCAB_SIZE,
    max_seq_len=max_seq_len_toy,
    dropout_rate=dropout_toy
)

# --- Prepare a single data sample ---
# Source: [SOS, 5, 3, 6, EOS]  (IDs: [1, 5+NUM_START_ID, 3+NUM_START_ID, 6+NUM_START_ID, 2])
# Target (input to decoder): [SOS, 6, 3, 5, EOS] (IDs: [1, 6+NUM_START_ID, 3+NUM_START_ID, 5+NUM_S
# Target (for loss):         [6, 3, 5, EOS, PAD] (IDs: [6+NUM_START_ID, 3+NUM_START_ID, 5+NUM_STAR

source_seq_toy = np.array([[SOS_ID, 5+NUM_START_ID, 3+NUM_START_ID, 6+NUM_START_ID, EOS_ID, PAD_ID
decoder_input_toy = np.array([[SOS_ID, 6+NUM_START_ID, 3+NUM_START_ID, 5+NUM_START_ID, EOS_ID, PAD
target_labels_toy = np.array([[6+NUM_START_ID, 3+NUM_START_ID, 5+NUM_START_ID, EOS_ID, PAD_ID, PAD

actual_src_len = 5
actual_tgt_len = 5 # SOS + reversed_seq + EOS

# --- Create Masks ---
# 1. Source Padding Mask (for Encoder Self-Attention)
src_padding_mask_toy = (source_seq_toy == PAD_ID)[:, np.newaxis, np.newaxis, :].astype(int)

# 2. Target Look-Ahead Mask (for Decoder Self-Attention)
tgt_seq_len_toy = decoder_input_toy.shape[1]
```

```
tgt_look_ahead_mask_triu = np.triu(np.ones((tgt_seq_len_toy, tgt_seq_len_toy)), k=1).astype(int)
tgt_padding_component = (decoder_input_toy == PAD_ID)[:, np.newaxis, :].astype(int)
tgt_look_ahead_mask_toy = np.maximum(tgt_look_ahead_mask_triu, tgt_padding_component)[:, np.newaxi

# 3. Encoder Output Padding Mask (for Decoder Encoder-Decoder Attention)
enc_out_padding_mask_toy = (source_seq_toy == PAD_ID)[:, np.newaxis, np.newaxis, :].astype(int)

print("--- Toy Example: Forward Pass & Loss ---")
print("Source Sequence (IDs):", source_seq_toy)
print("Decoder Input (IDs):", decoder_input_toy)
print("Target Labels (IDs for loss):", target_labels_toy)

# Forward pass
output_logits_toy = toy_transformer.forward(
    source_seq_toy,
    decoder_input_toy,
    src_padding_mask_toy,
    enc_out_padding_mask_toy,
    tgt_look_ahead_mask_toy,
    training=False # Or True if we had actual training
)

print("Output Logits shape:", output_logits_toy.shape) # (batch, target_seq_len, vocab_size)

# --- Loss Calculation: Cross-Entropy Loss ---
def cross_entropy_loss(logits, labels, pad_id):
    """
    Calculates cross-entropy loss, ignoring padding.
    Args:
        logits (np.ndarray): Output logits from the model (batch, seq_len, vocab_size).
        labels (np.ndarray): True token IDs (batch, seq_len).
        pad_id (int): Token ID for padding, to be ignored in loss.
    Returns:
        float: Average cross-entropy loss per non-padded token.
    """
    probs = softmax(logits, axis=-1)
    batch_size, seq_len, vocab_size = probs.shape

    # Select the probabilities corresponding to the true labels
    # This uses advanced indexing
    true_label_probs = probs[np.arange(batch_size)[:, np.newaxis],
                             np.arange(seq_len)[np.newaxis, :],
                             labels]

    # Avoid log(0) - clip probabilities
    true_label_probs = np.clip(true_label_probs, 1e-9, 1.0)
    log_probs = -np.log(true_label_probs)

    # Create a mask to ignore padding tokens in the loss
    non_pad_mask = (labels != pad_id).astype(float)

    # Apply mask and calculate sum of loss
    masked_log_probs = log_probs * non_pad_mask
    total_loss = np.sum(masked_log_probs)
    num_non_pad_tokens = np.sum(non_pad_mask)

    if num_non_pad_tokens == 0:
        return 0.0 # Avoid division by zero if all tokens are padding

    return total_loss / num_non_pad_tokens

loss = cross_entropy_loss(output_logits_toy, target_labels_toy, PAD_ID)
print(f"Calculated Cross-Entropy Loss: {loss:.4f}")

# In a real training loop, you would now:
# 1. Calculate gradients of 'loss' w.r.t. all learnable parameters in 'toy_transformer'.
#    (e.g., toy_transformer.source_embedding.embeddings, toy_transformer.final_linear_layer.weight
#     and all weights/biases within MHA, PFF, LayerNorm if they were fully implemented with learna
# 2. Update these parameters using an optimizer (e.g., Adam: params -= learning_rate * gradients).
```

10.3 Backpropagation and Optimization (Conceptual)

- **Backpropagation:** This is the algorithm used to compute gradients. It involves applying the chain rule recursively, starting from the loss function and going backward through each layer of the network. For a Transformer, this means calculating gradients for:
    - The final linear layer.
    - Each decoder block (FFN, encoder-decoder MHA, masked self-MHA, LayerNorms).

    - Each encoder block (FFN, self-MHA, LayerNorms).
    - Embedding layers.
      Manually deriving and implementing these for every matrix multiplication, addition, softmax, normalization, etc., is extremely complex and error-prone in NumPy.
- **Optimizers:** Once gradients are obtained, optimizers update the model parameters. Common optimizers include:
    - **SGD (Stochastic Gradient Descent):** `param = param - learning_rate * gradient`
    - **Adam:** A more sophisticated optimizer that adapts learning rates for each parameter, often leading to faster convergence.
      Implementing these also requires storing and updating additional variables (like momentum for Adam).

10.4 Inference (Prediction)

Once the model is trained, generating an output sequence for a new source sequence involves:

1. **Encode Source:** Pass the tokenized source sequence through the encoder to get `encoder_output`.
2. **Start Decoder Input:** Begin with a target sequence containing only the `SOS_ID`.
3. **Iterative Decoding:** Loop until `EOS_ID` is generated or a maximum length is reached:
   a. Create appropriate masks for the current target sequence.
   b. Pass the current `target_sequence`, `encoder_output`, and masks through the decoder and the final linear layer to get logits for the next token.
   c. Apply softmax to get probabilities.
   d. Select the next token (e.g., by taking the `argmax` of probabilities - greedy decoding, or by sampling - beam search is common for better quality).
   e. Append the predicted token to the `target_sequence`.
   f. If the predicted token is `EOS_ID`, stop.

10.5 Conclusion of the Series

Congratulations on reaching the end of this 10-part journey into building a Transformer from scratch with NumPy! We have covered an immense amount of ground:

- **Part 1: Introduction:** Set the stage for Transformers and NumPy.
- **Part 2: Scaled Dot-Product Attention:** Implemented the core attention mechanism.
- **Part 3: Multi-Head Attention:** Built a more powerful attention by combining multiple attention heads.
- **Part 4: Position-wise Feed-Forward Networks:** Added another key layer component.
- **Part 5: Positional Encoding:** Addressed the Transformer's lack of inherent sequence awareness.
- **Part 6: Encoder Block:** Assembled the first major building block of the Transformer.
- **Part 7: Full Transformer Encoder:** Stacked Encoder Blocks to create the complete encoder.
- **Part 8: Transformer Decoder Block:** Built the more complex decoder block with its two attention mechanisms.
- **Part 9: Complete Transformer Architecture:** Integrated the encoder, decoder, embeddings, and final output layer.
- **Part 10: Training and Conclusion:** Discussed the training process, loss functions, and the conceptual steps for making the model learn.

While our NumPy implementation focused on the forward pass and the architectural understanding, it provides a deep insight into the inner workings of one of the most influential neural network architectures today. The hands-on approach, even without a full training loop, demystifies many of the complex interactions within the model.

Key Takeaways from the Series:

- Transformers rely heavily on **self-attention** and **multi-head attention** to capture contextual relationships in sequences.
- **Positional encodings** are vital for injecting sequence order information.
- The **encoder-decoder architecture** is powerful for sequence-to-sequence tasks.
- **Residual connections** and **layer normalization** are crucial for training deep networks.
- **Masking** is essential for handling padding and ensuring auto-regressive behavior in the decoder.

From here, you are well-equipped to dive deeper into advanced Transformer variants, explore efficient implementations in frameworks like PyTorch or TensorFlow, and apply these concepts to various NLP tasks and beyond.

Thank you for following along, and happy coding!

Transformer from Scratch with NumPy - Tutorial Series

This repository contains a 10-part tutorial series on implementing the Transformer neural network architecture from scratch using only NumPy. The goal is to provide a clear, step-by-step guide with a focus on mathematical intuition and fundamental implementation details, avoiding high-level deep learning libraries.

Tutorial Parts

The series progressively builds the Transformer model, with each part focusing on specific components:

1. **Part 1: Introduction to Transformers and NumPy Setup**
   - Overview of the Transformer architecture.
   - Rationale for using NumPy.
   - Environment setup and basic NumPy operations.
2. **Part 2: Implementing Scaled Dot-Product Attention**
   - Explanation of the attention mechanism.
   - Detailed breakdown and NumPy implementation of Scaled Dot-Product Attention.
   - Masking.
3. **Part 3: Building Multi-Head Attention from Scratch**
   - Concept of Multi-Head Attention.
   - Implementation of the `MultiHeadAttention` class, combining multiple scaled dot-product attention heads.
4. **Part 4: Creating the Position-wise Feedforward Layer**
   - Description of the Position-wise Feed-Forward Network (FFN).
   - NumPy implementation of the `PositionwiseFeedForward` class.
5. **Part 5: Adding Positional Encoding**
   - Necessity of positional information in Transformers.
   - Implementation of sinusoidal positional encoding.
6. **Part 6: Constructing a Transformer Encoder Block**

- Architecture of a single Encoder Block.
- Implementation of Layer Normalization, Dropout (conceptual), and the `EncoderBlock` class, integrating Multi-Head Attention and FFN with residual connections.

7. **Part 7: Implementing the Full Transformer Encoder**
    - Stacking multiple Encoder Blocks.
    - Integrating input embeddings and positional encoding.
    - Implementation of the `TransformerEncoder` class.
8. **Part 8: Implementing the Transformer Decoder Block**
    - Architecture of a single Decoder Block (Masked Multi-Head Self-Attention, Encoder-Decoder Attention, FFN).
    - Implementation of the `DecoderBlock` class and look-ahead mask creation.
9. **Part 9: Building the Complete Transformer Architecture**
    - Assembling the full Transformer model: Encoder, Decoder, Embedding Layer, and Final Linear Layer.
    - Comprehensive mask creation.
    - Implementation of the full `Transformer` class.
10. **Part 10: Putting It All Together: Training on a Toy Dataset and Conclusion**
    - Conceptual discussion of the training loop, loss calculation (cross-entropy), backpropagation, and optimization.
    - Forward pass example on a toy dataset (e.g., sequence reversal).
    - Series conclusion and further learning.

How to Use

Each part is a self-contained Markdown file (`.md`) that includes explanations, mathematical formulas, and NumPy code snippets. You can follow the parts sequentially to build up your understanding and implementation. The code examples are designed to be run in a Python environment with NumPy installed.

Enjoy learning about Transformers!
