To dry-run the provided code for the `MultiHeadAttention` class, we'll trace the execution step-by-step with proper calculations, using the input tensor provided. The dry run will include detailed calculations and comments explaining each step, as requested. The comments will be added at the end of the code to summarize the dry run.

The code implements a Multi-Head Attention mechanism, a core component of Transformer models, using PyTorch. We'll assume the `nn.Module`, `nn.Linear`, `nn.Dropout`, and `torch` imports are available. Since the code uses `torch.manual_seed(123)` for reproducibility, we'll ensure calculations reflect this setting where applicable.

Code Overview

- **Input Tensor**: A batch of shape `(2, 3, 6)` (2 identical sequences, each with 3 tokens and 6 dimensions).
- **MultiHeadAttention Parameters**:
    - `d_in = 6` (input dimension)
    - `d_out = 6` (output dimension)
    - `context_length = 3` (sequence length)
    - `dropout = 0.0` (no dropout)
    - `num_heads = 2` (two attention heads)
    - `qkv_bias = False` (no bias in linear layers)
- **Goal**: Compute the output `context_vecs` and its shape, with a detailed dry run.

Step-by-Step Dry Run

1. Input Tensor and Initialization

```
torch.manual_seed(123)
inputs = torch.tensor(
    [[0.43, 0.15, 0.89, 0.55, 0.87, 0.66],  # Row 1
     [0.57, 0.85, 0.64, 0.22, 0.58, 0.33],  # Row 2
     [0.77, 0.25, 0.10, 0.05, 0.80, 0.55]]  # Row 3
)
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)  # Output: torch.Size([2, 3, 6])
```

- **Calculation**:
    - The `inputs` tensor is of shape `(3, 6)` (3 tokens, each with 6 features).
    - `torch.stack((inputs, inputs), dim=0)` creates a batch of shape `(2, 3, 6)` by stacking two identical `inputs` tensors along the batch dimension.
    - `batch_size = 2`, `context_length = 3`, `d_in = 6`.
- **Dry Run**:
    - Input batch shape: `(2, 3, 6)`.
    - Each sequence in the batch is identical to `inputs`.

2. MultiHeadAttention Initialization

```
batch_size, context_length, d_in = batch.shape
d_out = 6
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
```

- **Parameters**:
    - `d_in = 6`, `d_out = 6`, `context_length = 3`, `dropout = 0.0`, `num_heads = 2`, `qkv_bias = False`.
- **Assertions**:
    - Check: `d_out % num_heads == 0` → `6 % 2 = 0` (valid).
- **Attributes**:
    - `self.d_out = 6`.
    - `self.num_heads = 2`.
    - `self.head_dim = d_out // num_heads = 6 // 2 = 3`.
    - Linear layers:
        - `W_query`: `nn.Linear(6, 6, bias=False)` (maps input dim 6 to output dim 6).
        - `W_key`: `nn.Linear(6, 6, bias=False)`.
        - `W_value`: `nn.Linear(6, 6, bias=False)`.
        - `out_proj`: `nn.Linear(6, 6)` (combines head outputs, bias=True by default in PyTorch).
    - `self.dropout = nn.Dropout(0.0)` (no dropout applied).
    - Causal mask: `self.mask = torch.triu(torch.ones(3, 3), diagonal=1)`.

    ```
    # mask:
    tensor([[0., 1., 1.],
            [0., 0., 1.],
            [0., 0., 0.]])
    ```

- `torch.triu` creates an upper triangular matrix with 1s above the diagonal, 0s elsewhere.
- This mask ensures that each token only attends to itself and previous tokens (causal attention).

- **Dry Run**:
  - `head_dim = 3`.
  - Linear layers are initialized with random weights (seeded by `torch.manual_seed(123)`).
  - Causal mask is a `(3, 3)` tensor for a sequence length of 3.

3. Forward Pass: Input Processing

```
def forward(self, x):
    b, num_tokens, d_in = x.shape
```

- **Input**: `x = batch` with shape `(2, 3, 6)`.
- **Extract Dimensions**:
  - `b = 2` (batch size).
  - `num_tokens = 3` (sequence length).
  - `d_in = 6` (input dimension).

4. Compute Queries, Keys, Values

```
 keys = self.W_key(x)  # Shape: (2, 3, 6)
queries = self.W_query(x)
values = self.W_value(x)
```

- **Linear Transformations**:
  - Each linear layer (`W_key`, `W_query`, `W_value`) maps input `(2, 3, 6)` to output `(2, 3, 6)`.
  - Since `qkv_bias = False`, the transformation is `y = x @ W^T`, where `W` is a `(6, 6)` weight matrix.
- **Weights Initialization**:
  - With `torch.manual_seed(123)`, PyTorch's `nn.Linear` initializes weights using a uniform distribution (Kaiming initialization by default, adjusted for no bias).
  - For simplicity, let's denote the weight matrices as `W_key`, `W_query`, `W_value`, each of shape `(6, 6)`. Exact values depend on the random seed, but we'll assume they're initialized consistently.
- **Calculation (Simplified)**:
  - For a single token `x_i` (shape `(6,)`) in the sequence:
    - `keys_i = x_i @ W_key^T` (output shape `(6,)`).
    - Similarly for `queries_i` and `values_i`.
  - Applied across the batch and sequence: `keys`, `queries`, `values` each have shape `(2, 3, 6)`.
- **Dry Run**:
  - `keys.shape = (2, 3, 6)`.
  - `queries.shape = (2, 3, 6)`.
  - `values.shape = (2, 3, 6)`.

5. Reshape for Multi-Head Attention

```
 keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)  # (2, 3, 2, 3)
values = values.view(b, num_tokens, self.num_heads, self.head_dim)
queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)
```

- **Reshaping**:
  - `self.num_heads = 2`, `self.head_dim = 3`.
  - Original shape: `(2, 3, 6)` (batch, tokens, d_out).
  - New shape: `(2, 3, 2, 3)` (batch, tokens, heads, head_dim).
  - The `d_out = 6` dimension is split into `num_heads = 2` heads, each with `head_dim = 3` dimensions.
- **Dry Run**:
  - For each tensor (`keys`, `queries`, `values`):
    - Reshape splits the last dimension (6) into `(2, 3)` for 2 heads, each with 3 dimensions.
    - Shape becomes `(2, 3, 2, 3)`.

6. Transpose for Attention

```
 keys = keys.transpose(1, 2)  # (2, 2, 3, 3)
queries = queries.transpose(1, 2)
values = values.transpose(1, 2)
```

- **Transpose**:
  - Swap dimensions 1 (tokens) and 2 (heads).
  - Shape changes from `(2, 3, 2, 3)` to `(2, 2, 3, 3)` (batch, heads, tokens, head_dim).
- **Purpose**:
  - This aligns the tensors for batched matrix multiplication per head.
- **Dry Run**:

- `keys.shape = (2, 2, 3, 3)`.
- `queries.shape = (2, 2, 3, 3)`.
- `values.shape = (2, 2, 3, 3)`.

7. Scaled Dot-Product Attention

```
attn_scores = queries @ keys.transpose(2, 3)  # (2, 2, 3, 3)
```

- **Matrix Multiplication**:
  - `keys.transpose(2, 3)` transposes the last two dimensions: `(2, 2, 3, 3)` → `(2, 2, 3, 3)` (batch, heads, tokens, head_dim).
  - `queries @ keys.transpose(2, 3)`:
    - For each batch and head, compute dot product: `(3, 3) @ (3, 3) → (3, 3)`.
    - Resulting shape: `(2, 2, 3, 3)` (batch, heads, query_tokens, key_tokens).
- **Calculation (Example for One Head)**:
  - For batch 1, head 1, let `Q` and `K` be the query and key matrices for the 3 tokens, each of shape `(3, 3)`.
  - `attn_scores = Q @ K^T` produces a `(3, 3)` matrix where `attn_scores[i,j]` is the dot product of query token `i` and key token `j`.
- **Dry Run**:
  - `attn_scores.shape = (2, 2, 3, 3)`.

8. Apply Causal Mask

```
mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
attn_scores.masked_fill_(mask_bool, -torch.inf)
```

- **Mask**:
  - `self.mask` is a `(3, 3)` upper triangular matrix with 1s above the diagonal:

    ```
    tensor([[0., 1., 1.],
            [0., 0., 1.],
            [0., 0., 0.]])
    ```

  - `mask_bool = self.mask.bool()[:3, :3]` converts to boolean:

    ```
    tensor([[False, True,  True],
            [False, False, True],
            [False, False, False]])
    ```

  - `attn_scores.masked_fill_(mask_bool, -torch.inf)` sets positions where `mask_bool = True` to `-inf`.
- **Effect**:
  - For each head and batch, the attention scores matrix `(3, 3)` has `-inf` in positions where future tokens would be attended (ensuring causal attention).
  - Example for one head:

    ```
    attn_scores = [[s11, s12, s13],
                   [s21, s22, s23],
                   [s31, s32, s33]]
    # After masking:
    attn_scores = [[s11, -inf, -inf],
                   [s21, s22,  -inf],
                   [s31, s32,  s33]]
    ```

- **Dry Run**:
  - `attn_scores` now has `-inf` in upper triangular positions, enforcing that each token attends only to itself and previous tokens.

9. Compute Attention Weights

```
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
attn_weights = self.dropout(attn_weights)
```

- **Scaling**:
  - `keys.shape[-1] = head_dim = 3`.
  - Scale `attn_scores` by `1 / sqrt(3) ≈ 1 / 1.732 ≈ 0.577`.
  - `attn_scores = attn_scores / 3**0.5`.
- **Softmax**:
  - Apply softmax over the last dimension (key_tokens) to get attention weights.
  - For each head and batch, transform the `(3, 3)` matrix:
    - Positions with `-inf` become 0 after softmax.
    - Example for one head:

```
# Before softmax (after scaling):
attn_scores = [[s11', -inf, -inf],
               [s21', s22', -inf],
               [s31', s32', s33']]
# After softmax (dim=-1):
attn_weights = [[1, 0, 0],
                [softmax(s21', s22'), 0],
                [softmax(s31', s32', s33')]]
```

        - For the first row, only the first position is non- `-inf` , so weight = 1.
        - For the second row, softmax normalizes over the first two positions.
        - For the third row, softmax normalizes over all three positions.
- **Dropout**:
    - `self.dropout = nn.Dropout(0.0)` , so no dropout is applied.
    - `attn_weights` remains unchanged.
- **Dry Run**:
    - `attn_weights.shape = (2, 2, 3, 3)` .
    - Each `(3, 3)` matrix is lower triangular (with zeros above the diagonal after softmax).

10. Compute Context Vectors

```
context_vec = (attn_weights @ values).transpose(1, 2)  # (2, 3, 2, 3)
```

- **Matrix Multiplication**:
    - `attn_weights @ values` :
        - `attn_weights` : `(2, 2, 3, 3)` (batch, heads, query_tokens, key_tokens).
        - `values` : `(2, 2, 3, 3)` (batch, heads, tokens, head_dim).
        - Result: `(2, 2, 3, 3)` (batch, heads, query_tokens, head_dim).
    - For each batch and head, `attn_weights @ values` computes a weighted sum of `values` for each query token.
- **Transpose**:
    - `transpose(1, 2)` : `(2, 2, 3, 3)` → `(2, 3, 2, 3)` (batch, tokens, heads, head_dim).
- **Dry Run**:
    - `context_vec.shape = (2, 3, 2, 3)` .

11. Combine Heads

```
context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)  # (2, 3, 6)
```

- **Reshape**:
    - `context_vec` : `(2, 3, 2, 3)` → `(2, 3, 6)` .
    - Combine `num_heads = 2` and `head_dim = 3` into `d_out = 2 * 3 = 6` .
    - `contiguous()` ensures the tensor is stored in a contiguous memory block for the view operation.
- **Dry Run**:
    - `context_vec.shape = (2, 3, 6)` .

12. Final Projection

```
context_vec = self.out_proj(context_vec)  # (2, 3, 6)
return context_vec
```

- **Linear Layer**:
    - `self.out_proj` : `nn.Linear(6, 6)` (with bias by default).
    - Applies a linear transformation to each token's vector: `(2, 3, 6) → (2, 3, 6)` .
- **Dry Run**:
    - `context_vec.shape = (2, 3, 6)` (unchanged).
    - The output is the final context vector for each token in the sequence.

13. Output

```
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

- **Shape**: `(2, 3, 6)` (batch, tokens, d_out).
- **Values**:
    - Exact values depend on the randomly initialized weights (`W_query` , `W_key` , `W_value` , `out_proj`) set by `torch.manual_seed(123)` .
    - Computing the exact numerical output requires the weight matrices, which are not provided in the code. However, the process is deterministic given the seed.
- **Sample Output (Hypothetical)**:
    - Since we can't compute exact values without weights, the output is a tensor of shape `(2, 3, 6)` .
    - Example structure (values are illustrative):

```
tensor([[[...], [...], [...]],  # Batch 1: 3 tokens, each with 6 dims
        [[...], [...], [...]]]) # Batch 2: identical due to identical input
```

- The actual `print(context_vecs)` would show a `(2, 3, 6)` tensor with values transformed through the attention mechanism.

Final Code with Dry Run Comments

```
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads

        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length), diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape

        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)

        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        attn_scores = queries @ keys.transpose(2, 3)

        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2)
        context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)
        context_vec = self.out_proj(context_vec)

        return context_vec

torch.manual_seed(123)

# Define the tensor with 3 rows and 6 columns
inputs = torch.tensor(
    [[0.43, 0.15, 0.89, 0.55, 0.87, 0.66],  # Row 1
     [0.57, 0.85, 0.64, 0.22, 0.58, 0.33],  # Row 2
```

```
     [0.77, 0.25, 0.10, 0.05, 0.80, 0.55]]  # Row 3
)

batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)  # torch.Size([2, 3, 6])

batch_size, context_length, d_in = batch.shape
d_out = 6
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)

# Dry Run Comments:
# 1. Input Setup:
#    - inputs: tensor of shape (3, 6) with 3 tokens, each with 6 dimensions.
#    - batch: stacked to (2, 3, 6) with two identical sequences.
#    - batch_size = 2, context_length = 3, d_in = 6, d_out = 6, num_heads = 2.
#
# 2. MultiHeadAttention Initialization:
#    - head_dim = d_out // num_heads = 6 // 2 = 3.
#    - W_query, W_key, W_value: nn.Linear(6, 6, bias=False) initialized with seed 123.
#    - out_proj: nn.Linear(6, 6, bias=True).
#    - mask: (3, 3) upper triangular matrix with 1s above diagonal, 0s elsewhere.
#    - dropout: 0.0 (no dropout).
#
# 3. Forward Pass:
#    - Input x: shape (2, 3, 6).
#    - Compute keys, queries, values: each shape (2, 3, 6) via linear transformations.
#    - Reshape to (2, 3, 2, 3) to split d_out=6 into 2 heads, each with head_dim=3.
#    - Transpose to (2, 2, 3, 3) for batched attention per head.
#
# 4. Attention Scores:
#    - attn_scores = queries @ keys.transpose(2, 3): shape (2, 2, 3, 3).
#    - For each batch and head, compute (3, 3) matrix of query-key dot products.
#
# 5. Causal Masking:
#    - mask_bool: (3, 3) boolean mask, True above diagonal.
#    - Set upper triangular positions in attn_scores to -inf to prevent attending to future tokens.
#
# 6. Attention Weights:
#    - Scale attn_scores by 1/sqrt(head_dim) = 1/sqrt(3) ≈ 0.577.
#    - Apply softmax over last dimension: converts scores to probabilities, with -inf → 0.
#    - Dropout (0.0): no change.
#    - attn_weights: shape (2, 2, 3, 3), lower triangular structure.
#
# 7. Context Vectors:
#    - attn_weights @ values: shape (2, 2, 3, 3), weighted sum of values per head.
#    - Transpose to (2, 3, 2, 3).
#
# 8. Combine Heads:
#    - Reshape to (2, 3, 6) by combining num_heads=2 and head_dim=3.
#
# 9. Final Projection:
#    - out_proj: linear transformation (2, 3, 6) → (2, 3, 6).
#
# 10. Output:
#    - context_vecs: shape (2, 3, 6).
#    - Values depend on random weights (seed 123), not computed explicitly here.
#    - Each token's output is a 6-dim vector, contextualized via attention.
```

Final Answer

- **Output Shape**: `context_vecs.shape = (2, 3, 6)`.

- **Output Values**: The exact numerical values of `context_vecs` depend on the randomly initialized weights of the linear layers, set by `torch.manual_seed(123)`. Without explicit weight values, we can't compute the precise tensor, but the shape and process are as described.
- **Dry Run**: Provided in comments at the end of the code, detailing each step's operations and shapes.

If you need the exact numerical output, you'd need to run the code with PyTorch to get the weight values initialized with the seed. Alternatively, if you have specific weight matrices or want a more detailed numerical example for a single head, let me know!