```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# Класс ScaledDotProductAttention (без изменений)
class ScaledDotProductAttention(nn.Module):
    def __init__(self, dropout=0.1):
        super(ScaledDotProductAttention, self).__init__()
        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, mask=None):
        # Вычисление скалярного произведения между запросами и ключами
        scores = torch.matmul(query, key.transpose(-2, -1)) / torch.sqrt(torch.tensor(query.size(-1), dtype=torch.float32))

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Применение softmax для получения весов внимания
        attn = F.softmax(scores, dim=-1)
        attn = self.dropout(attn)

        # Суммирование значений с учетом весов внимания
        output = torch.matmul(attn, value)

        return output, attn

# Обновлённый класс MultiHeadAttention (с исправленной маской)
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, d_model, dropout=0.1):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "Размерность модели должна делиться на количество голов"

        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        self.linear_q = nn.Linear(d_model, d_model)
        self.linear_k = nn.Linear(d_model, d_model)
        self.linear_v = nn.Linear(d_model, d_model)

        self.attention = ScaledDotProductAttention(dropout)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)
```

```python
    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # Линейные преобразования и разделение на головы
        query = self.linear_q(query).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        key = self.linear_k(key).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        value = self.linear_v(value).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        if mask is not None:
            # Преобразуем маску из (batch_size, seq_length) в (batch_size, 1, 1, seq_length)
            mask = mask.unsqueeze(1).unsqueeze(2)

        # Вычисление внимания
        out, attn = self.attention(query, key, value, mask)

        # Объединение голов и линейное преобразование
        out = out.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)
        out = self.linear_out(out)
        out = self.dropout(out)

        # Добавление residual связи и нормализация
        out = self.layer_norm(out + query.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k))

        return out, attn

# Пример использования
# Параметры
batch_size = 2
seq_length = 5
d_model = 16
num_heads = 4

# Создание случайных тензоров для запроса, ключа и значения
query = torch.randn(batch_size, seq_length, d_model)
key = torch.randn(batch_size, seq_length, d_model)
value = torch.randn(batch_size, seq_length, d_model)

# Создание маски (например, для игнорирования паддинга)
mask = torch.ones(batch_size, seq_length)  # Здесь все элементы актуальны

# Инициализация слоя Multi-Head Attention
mha = MultiHeadAttention(num_heads=num_heads, d_model=d_model)

# Прямой проход
output, attn = mha(query, key, value, mask)

print("Выход:", output.shape)        # Ожидается (batch_size, seq_length, d_model)
print("Веса внимания:", attn.shape)  # Ожидается (batch_size, num_heads, seq_length, seq_length)
```

```
Выход: torch.Size([2, 5, 16])
Веса внимания: torch.Size([2, 4, 5, 5])
```

```python
print("Выход:", output.shape)        # Ожидается (batch_size, seq_length, d_model)
print("Веса внимания:", attn.shape)  # Ожидается (batch_size, num_heads, seq_length, seq_length)
print("Query shape:", query.shape)
print("Key shape:", key.shape)
print("Value shape:", value.shape)
print("Mask shape:", mask.shape if mask is not None else "None")
```

```
Выход: torch.Size([2, 5, 16])
Веса внимания: torch.Size([2, 4, 5, 5])
Query shape: torch.Size([2, 5, 16])
Key shape: torch.Size([2, 5, 16])
Value shape: torch.Size([2, 5, 16])
Mask shape: torch.Size([2, 5])
```