2023.11.09 - homework 2

November 9, 2023

0.0.1 2023.11.09 - Introduction to Transformers Continued | Homework 2

In this exercise, you will implement a masked scaled dot-product attention, which allows the model to focus on different parts of the input sequence.

Base your code on the following skeleton code that we provide:

0.0.2 Exercise 1 - Scaled Dot-Product Attention:

Implement the scaled dot-product attention function, which is defined as `Attention(Q, K, V) = softmax(QK^T / sqrt(d_k))V`, where Q, K, V are queries, keys, and values, respectively, and `d_k` is the dimensionality of the keys.

[78]:
```
from torch import Tensor
# TODO: Import necessary dependencies
```

[74]:
```
def scaled_dot_product_attention(Q: Tensor, K: Tensor, V: Tensor) -> (Tensor,␣
 ↪Tensor):
    """
    Computes the scaled dot-product attention.
    The function takes three inputs: Q (query), K (key), V (value).
    It computes the dot products of the query with all keys, divides each by␣
 ↪sqrt(d_k), and applies a softmax function to obtain the weights on the␣
 ↪values.

    Parameters:
    - Q: Tensor of shape (batch_size, block_size, d_k)
    - K: Tensor of shape (batch_size, block_size, d_k)
    - V: Tensor of shape (batch_size, block_size, d_k)

    Returns:
    - output: Tensor of shape (batch_size, block_size, d_k)
        The resulting tensor after applying the attention mechanism.
    - attention_weights: Tensor of shape (batch_size, block_size, block_size)
        The attention weights after applying softmax.
    """
    pass
```

Run Exercise 1 Run this cell to evaluate your implementation.

[59]:
```
"""
Expected Output:
============ Scaled dot-product attention
Attention Output:
tensor([[[ 0.2957,  0.3332,  0.0674,  0.2081], # these numbers will be␣
 ↪different, but the dimensions should be the same
         [ 0.2628,  0.4036,  0.0423,  0.2148],
         [ 0.3083,  0.4486,  0.2576,  0.2905]],

        [[-0.1859, -0.8284, -0.5916, -1.0532],
         [-0.1266, -0.8076, -0.5838, -1.1651],
         [ 0.1222, -0.4625, -0.4709, -0.1724]]])

Attention Weights:
tensor([[[0.4537, 0.3877, 0.1586],
         [0.4461, 0.3433, 0.2105],
         [0.2678, 0.3638, 0.3684]],

        [[0.5597, 0.1483, 0.2920],
         [0.5515, 0.0702, 0.3784],
         [0.1524, 0.7299, 0.1178]]])
"""
print("============ Scaled dot-product attention")
torch.manual_seed(0)      # to keep results consistent between runs
Q = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4
K = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4
V = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4

output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Attention Output:\n{output}\n")
print(f"Attention Weights:\n{weights}")
```

```
============ Scaled dot-product attention
Attention Output:
tensor([[[ 0.2957,  0.3332,  0.0674,  0.2081],
         [ 0.2628,  0.4036,  0.0423,  0.2148],
         [ 0.3083,  0.4486,  0.2576,  0.2905]],

        [[-0.1859, -0.8284, -0.5916, -1.0532],
         [-0.1266, -0.8076, -0.5838, -1.1651],
         [ 0.1222, -0.4625, -0.4709, -0.1724]]])

Attention Weights:
tensor([[[0.4537, 0.3877, 0.1586],
         [0.4461, 0.3433, 0.2105],
         [0.2678, 0.3638, 0.3684]],

        [[0.5597, 0.1483, 0.2920],
         [0.5515, 0.0702, 0.3784],
         [0.1524, 0.7299, 0.1178]]])
```

0.0.3 Excercise 2 - Add a mask to the attention mechanism:

The next step is to add a masking functionality to the attention mechanism. A mask allows the model to selectively ignore certain positions within the sequence, used to handle variable sequence lengths or to prevent the model from attending to future positions in the a decoder model.

Note: The mask should have the same batch size and block size as the queries and should contain ones for positions that should be attended to and zeros for masked positions. Before the softmax step, masked positions are filled with large negative values (e.g. -torch.inf), effectively removing these positions from consideration in the attention weights.

Feel free to copy & paste your implementation from Excercise 1.

[79]:
```
def scaled_dot_product_attention(Q: Tensor, K: Tensor, V: Tensor, mask:␣
↪Tensor=None) -> (Tensor, Tensor):
    """
    Computes the scaled dot-product attention.
    The function takes three inputs: Q (query), K (key), V (value).
    It computes the dot products of the query with all keys, divides each by␣
↪sqrt(d_k), and applies a softmax function to obtain the weights on the␣
↪values.

    Parameters:
    - Q: Tensor of shape (batch_size, block_size, d_k)
    - K: Tensor of shape (batch_size, block_size, d_k)
    - V: Tensor of shape (batch_size, block_size, d_k)
    - mask: Optional tensor of shape (batch_size, 1, block_size).
            The mask should contain 1s for positions to attend to and 0s for␣
↪masked positions.
            Masked positions are filled with large negative values before the␣
↪softmax step, effectively excluding them from consideration in the attention␣
↪weights.

    Returns:
    - output: Tensor of shape (batch_size, block_size, d_k)
        The resulting tensor after applying the attention mechanism.
    - attention_weights: Tensor of shape (batch_size, block_size, block_size)
        The attention weights after applying softmax.
    """
    pass
```

Run Exercise 2 Run these cells to evaluate your implementation.

[57]:
```
"""
Expected Output:
```

```
============ Masked scaled dot-product attention
Attention Output:
tensor([[[ 0.2957,  0.3332,  0.0674,  0.2081],
         [ 0.2628,  0.4036,  0.0423,  0.2148],
         [ 0.3083,  0.4486,  0.2576,  0.2905]],

        [[-0.5912, -1.0937, -0.6829, -0.9884],
         [-0.6886, -1.1866, -0.7139, -1.1376],
         [ 0.0322, -0.4995, -0.4843, -0.0339]]])

Attention Weights:
tensor([[[0.4537, 0.3877, 0.1586],
         [0.4461, 0.3433, 0.2105],
         [0.2678, 0.3638, 0.3684]],

        [[0.7905, 0.2095, 0.0000],
         [0.8871, 0.1129, 0.0000],
         [0.1727, 0.8273, 0.0000]]])
"""
print("============ Masked scaled dot-product attention")
torch.manual_seed(0)      # to keep results consistent between runs
Q = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4
K = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4
V = torch.randn(2, 3, 4)  # batch size of 2, block size of 3, and d_k of 4

mask = torch.ones_like(Q[:, :, 0]).unsqueeze(1)  # Mask of shape (batch size, 1, block size)
mask[1, 0, 2:] = 0 # Mask last token of the second block

output, weights = scaled_dot_product_attention(Q, K, V, mask)
print(f"Attention Output:\n{output}\n")
print(f"Attention Weights:\n{weights}")
```

```
============ Masked scaled dot-product attention
Attention Output:
tensor([[[ 0.2957,  0.3332,  0.0674,  0.2081],
         [ 0.2628,  0.4036,  0.0423,  0.2148],
         [ 0.3083,  0.4486,  0.2576,  0.2905]],

        [[-0.5912, -1.0937, -0.6829, -0.9884],
         [-0.6886, -1.1866, -0.7139, -1.1376],
         [ 0.0322, -0.4995, -0.4843, -0.0339]]])

Attention Weights:
tensor([[[0.4537, 0.3877, 0.1586],
         [0.4461, 0.3433, 0.2105],
         [0.2678, 0.3638, 0.3684]],
[[0.7905, 0.2095, 0.0000],
 [0.8871, 0.1129, 0.0000],
 [0.1727, 0.8273, 0.0000]]])
```