

## Exercise 1 - Dot-Product Attention

You are given a set of vectors

$$\mathbf{h}_1 = (1, 2, 3)^\top, \quad \mathbf{h}_2 = (1, 2, 1)^\top, \quad \mathbf{h}_3 = (0, 1, -1)^\top$$

and an alignment source vector  $\mathbf{s} = (1, 2, 1)^\top$ . Compute the resulting dot-product attention weights  $\alpha_i$  for  $i = 1, 2, 3$  and the resulting context vector  $\mathbf{c}$ .

### Solution

First we compute the dot products between  $\mathbf{s}$  and the  $\mathbf{h}_i$  and apply softmax resulting in:

$$\mathbf{a} = \text{Softmax} \left( \begin{pmatrix} 1 & 1 & 0 \\ 2 & 2 & 1 \\ 3 & 1 & -1 \end{pmatrix}^\top \cdot \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \right) \approx \begin{pmatrix} 0.88 \\ 0.12 \\ 0.00 \end{pmatrix}$$

The resulting context vector is then computed as a weighted sum of the  $\mathbf{h}_i$ :

$$\mathbf{c} = a_1 \mathbf{h}_1 + a_2 \mathbf{h}_2 + a_3 \mathbf{h}_3 \approx \begin{pmatrix} 1.00 \\ 2.00 \\ 2.76 \end{pmatrix}$$

A simple implementation yielding the solution is as follows:

```
import torch
h = torch.tensor([[1, 2, 3], [1, 2, 1], [0, 1, -1]])
s = torch.tensor([1, 2, 1])
a = torch.matmul(h, s).float()
a = torch.exp(a) / torch.sum(torch.exp(a)) # Softmax
c = a[0] * h[0] + a[1] * h[1] + a[2] * h[2]
print(c)
```

## Exercise 2 - Attention in Transformers

Transformers use a scaled dot product attention mechanism given by

$$\mathbf{C} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} \right) \mathbf{V},$$

where  $\mathbf{Q} \in \mathbb{R}^{n_q \times d_k}$ ,  $\mathbf{K} \in \mathbb{R}^{n_k \times d_k}$ ,  $\mathbf{V} \in \mathbb{R}^{n_k \times d_v}$ .

- Is the softmax function here applied row-wise or column-wise? What is the shape of the result?
- What is the value of  $d$ ? Why is it needed?
- What is the computational complexity of this attention mechanism? How many additions and multiplications are required? Assume the canonical matrix multiplication and not counting  $\exp(x)$  towards computational cost.
- In the masked variant of the module, a masking matrix is added before the softmax function is applied. What are its values and its shape? For simplicity, assume  $n_q = n_k$ .

**Solution**

- (a) The softmax function is applied row-wise and the shape of the result is  $n_q \times n_k$ . One way to see this is by looking at the shape of the dot product  $QK^\top$  which is  $n_q \times n_k$ . Each row represents the pre-softmax scores of all keys and a given query. Because we need to normalize our attention weights per query, the normalization happens along the rows.
- (b) The value of  $d$  is  $d_k$ . It is needed to scale the dot product so that the gradient of the softmax function does not vanish.
- (c) To obtain the computational complexity, let's look at all the operations individually:
- $QK^\top$  requires  $n_q n_k d_k$  multiplications and  $n_q n_k (d_k - 1)$  additions.
  - Dividing by  $\sqrt{d_k}$  needs to be carried out  $n_q n_k$  times.
  - Applying the softmax function can be implemented in  $n_q n_k$  divisions and  $n_q (n_k - 1)$  additions.
  - The final matrix multiplication requires  $n_q d_v n_k$  multiplications and  $n_q d_v (n_k - 1)$  additions.
- (d) The masking matrix is a triangular matrix with  $-\infty$  on its top right half. This results in softmax weights being 0 for all key-query combinations to which  $-\infty$  is added.

**Exercise 3 - Scaled Dot-Product Attention by Hand**

Consider the matrices  $Q$ ,  $K$ ,  $V$  given by

$$Q = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}, \quad K = \begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & 2 \\ 0 & 1 & -1 \end{bmatrix}.$$

Compute the context matrix  $C$  using the scaled dot product attention.

**Solution**

The resulting context matrix is given by:

$$C \approx \begin{pmatrix} 0.86 & 1.58 & -0.72 \\ 0.99 & 1.88 & -1.56 \end{pmatrix}$$

A simple implementation would look as follows:

```
import torch
Q = torch.tensor([[1, 2], [3, 1]]).float()
K = torch.tensor([[2, 1], [1, 1], [0, 1]]).float()
V = torch.tensor([[1, 2, -2], [1, 1, 2], [0, 1, -1]]).float()
d_k = torch.tensor(K.shape[1])
M = torch.matmul(Q, K.transpose(0, 1)) / torch.sqrt(d_k)
S = torch.exp(M) / torch.sum(torch.exp(M), dim=1).view(-1, 1)
torch.matmul(S, V)
```

Pytorch also provides a function for scaled dot product attention:

```
import torch.nn.functional as F
F.scaled_dot_product_attention(Q, K, V)
```