Deep learning

13.2. Attention Mechanisms

François Fleuret
`https://fleuret.org/dlc/`
Dec 20, 2020

The simplest form of attention is **content-based attention**. Given an "attention function"

$$a : \mathbb{R}^D \times \mathbb{R}^C \to \mathbb{R}$$

and model parameters

$$\theta \in \mathbb{R}^{T \times C}$$

this operation takes a "value" tensor as input

$$V \in \mathbb{R}^{S \times D}$$

and computes an output

$$Y \in \mathbb{R}^{T \times D}$$

with

$$\begin{aligned} \forall j = 1, \dots, T, \quad Y_j &= \sum_{i=1}^{S} \frac{\exp(a(V_i; \theta_j))}{\sum_{k=1}^{T} \exp(a(V_k; \theta_j))} V_i \\ &= \sum_{i=1}^{S} \operatorname{softmax}_i \left(a(V_i; \theta_j) \right) V_i . \end{aligned}$$

This differs from **context attention**, which, given two inputs: a “context” tensor

$$C \in \mathbb{R}^{T \times C}$$

and a “value” tensor

$$V \in \mathbb{R}^{S \times D}$$

computes a tensor

$$Y \in \mathbb{R}^{T \times D}$$

with

$$\forall j = 1, \dots, T, \; Y_j = \sum_{i=1}^{S} \operatorname{softmax}_i \left(a \left(C_j, V_i; \theta \right) \right) V_i.$$

When $C = V$, this is **self-attention**.

The most classical version of attention is a context-attention with a dot-product for attention function, as used by Vaswani et al. (2017) for their transformer models. We will come back to them.

Also, using the terminology of Graves et al. (2014), attention is an averaging of **values** associated to **keys** matching a **query**. Hence the keys used for computing attention and the values to average are different quantities.

With Q the tensor of row queries, K the keys, and V the values,

$$Q \in \mathbb{R}^{T \times D} \quad K \in \mathbb{R}^{T' \times D} \quad V \in \mathbb{R}^{T' \times D'},$$

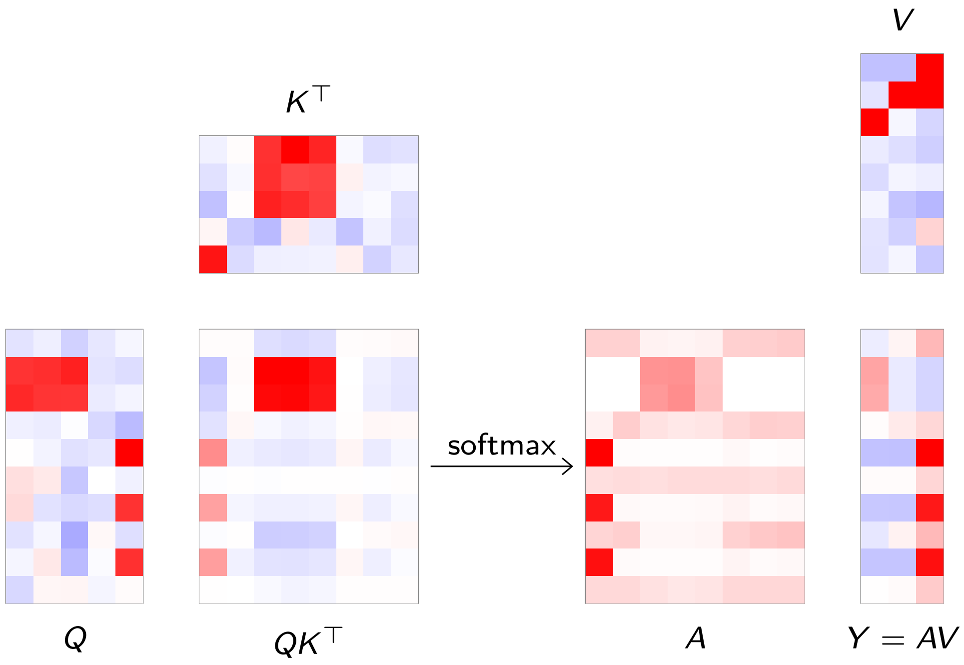
the result of the attention operation is

$$Y_j = \sum_i \frac{\exp(Q_j K_i^\top)}{\sum_r \exp(Q_j K_r^\top)} V_i,$$

or

$$Y = \underbrace{\text{softmax}\left(Q\,K^\top\right)}_{\text{Attention matrix } A} V.$$

The queries and keys have the same dimension D. There are as many keys T' as there are values. The result has as many rows T as there are queries, and they are of same dimension D' as the values.



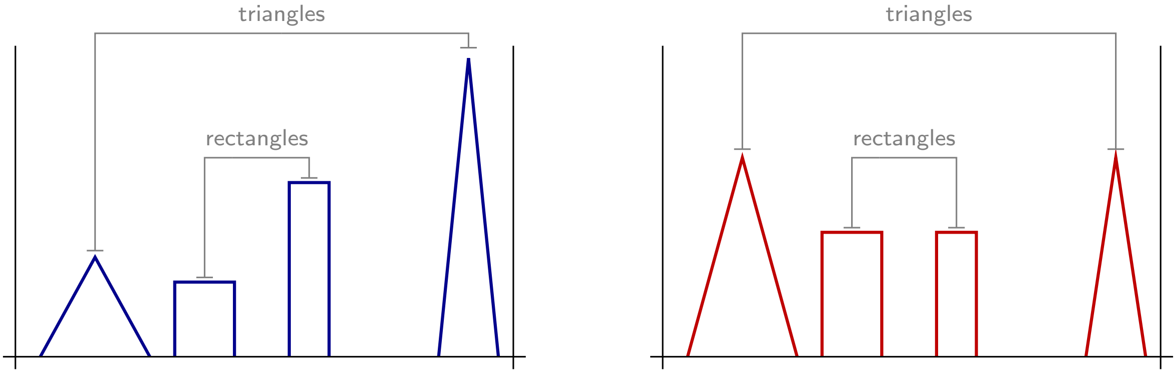
In the currently standard models for sequences, the queries, keys, and values are linear functions of the inputs.

Hence given three matrices $W_Q \in \mathbb{R}^{D \times C}$, $W_K \in \mathbb{R}^{D \times C'}$, and $W_V \in \mathbb{R}^{D' \times C'}$, and two input sequences $X \in \mathbb{R}^{T \times C}$, $X' \in \mathbb{R}^{T' \times C'}$, we have

$$\begin{cases} Q & = X\, W_Q^\top \in \mathbb{R}^{T \times D} \\ K & = X'\, W_V^\top \in \mathbb{R}^{T' \times D} \\ V & = X'\, W_V^\top \in \mathbb{R}^{T' \times D'} \end{cases}$$

As for context-attention, if $X = X'$ this is **self-attention**.

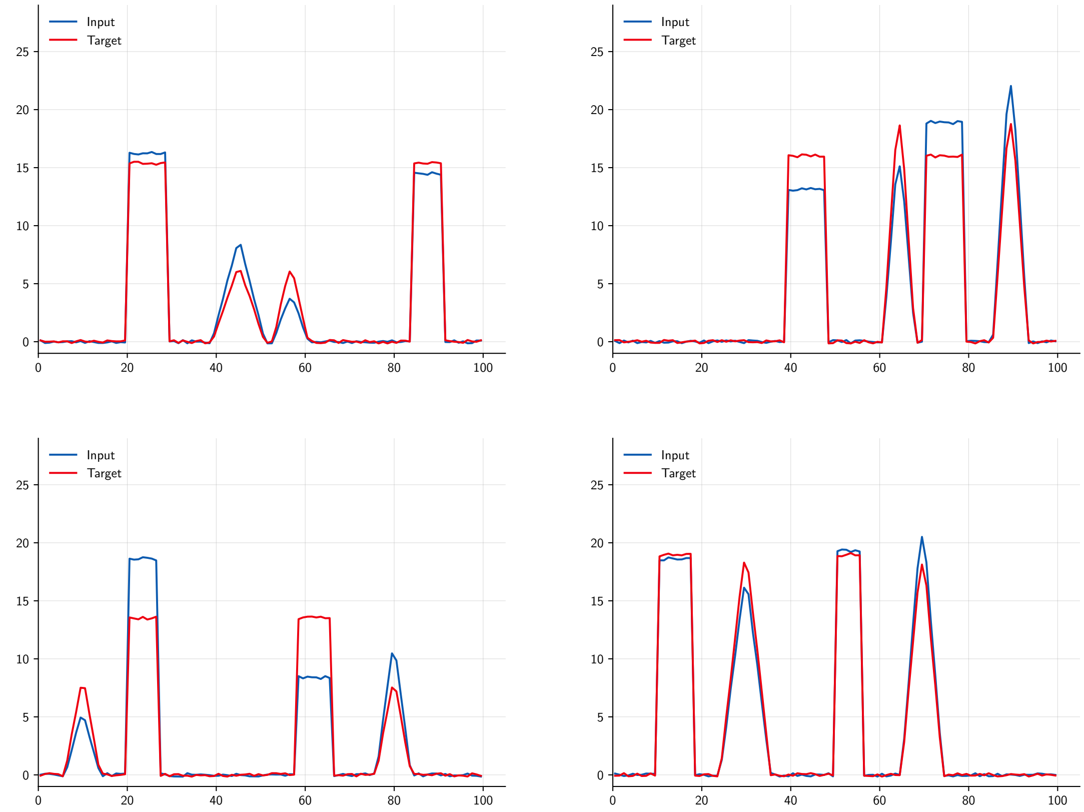
To illustrate the behavior of such an attention layer, we consider a toy problem with 1d sequences composed of two triangular and two rectangular patterns. The target averages the heights in each **pair of shapes**.



Input

Target

Some training examples.

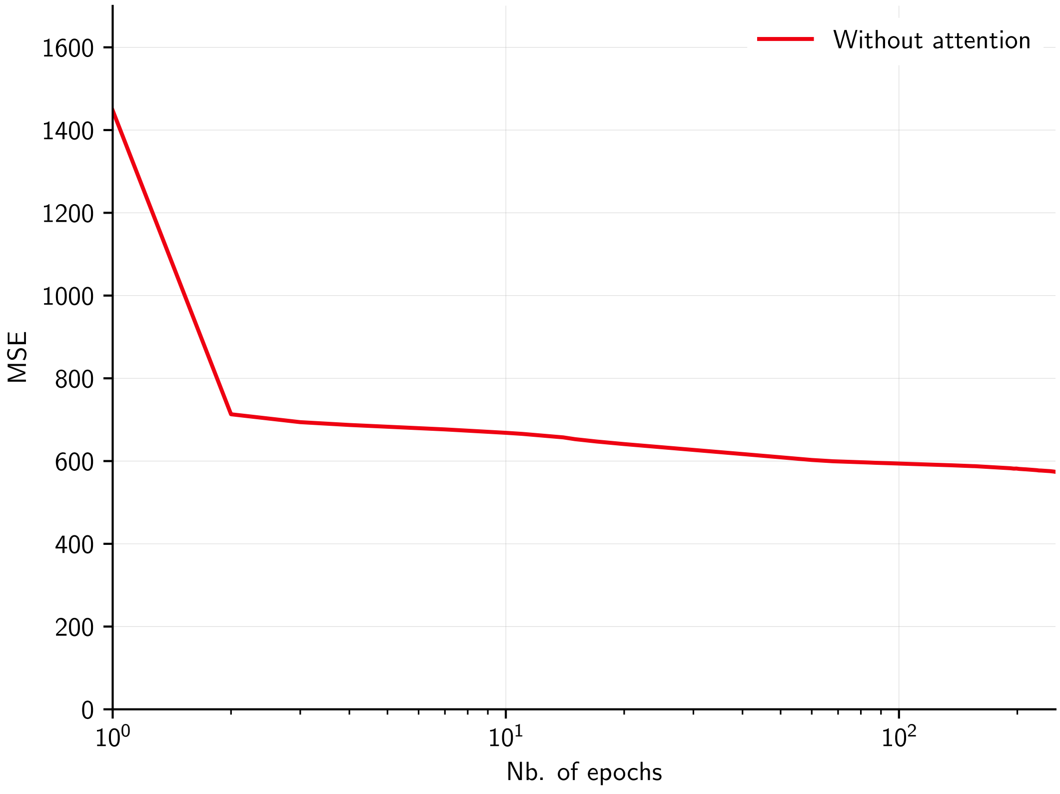


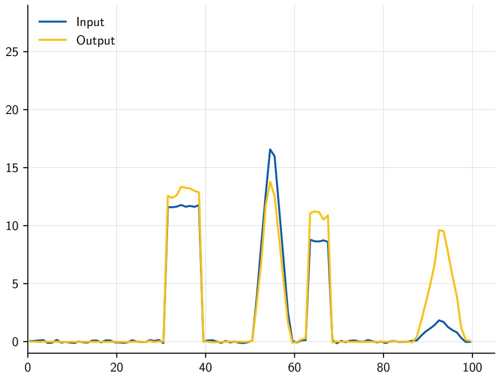
We test first a 1d convolutional network, with no attention mechanism.

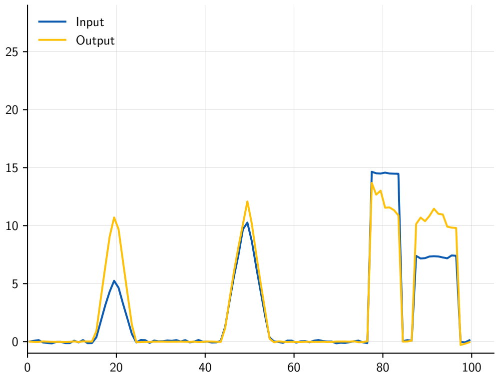
```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (1): ReLU()
  (2): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (3): ReLU()
  (4): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (5): ReLU()
  (6): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (7): ReLU()
  (8): Conv1d(64, 1, kernel_size=(5,), stride=(1,), padding=(2,))
)

nb_parameters 62337
```

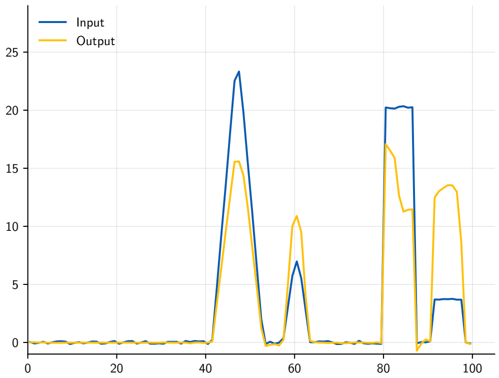
Training is done with the MSE loss and Adam.

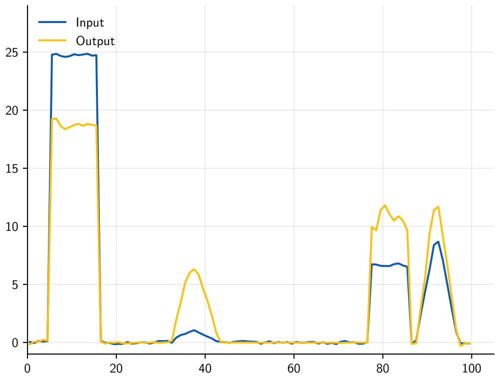
```
batch_size = 100

optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
mse_loss = nn.MSELoss()


mu, std = train_input.mean(), train_input.std()

for e in range(args.nb_epochs):

    for input, targets in zip(train_input.split(batch_size),
                              train_targets.split(batch_size)):


        output = model((input - mu) / std)
        loss = mse_loss(output, targets)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```











The poor performance of this model is not surprising given its inability to channel information from "far away" in the signal. Using more layers, global channel averaging, or fully connected layers could possibly solve the problem.

However it is more natural to equip the model with the ability to combine information from parts of the signal that it actively identifies as relevant.

This is exactly what an attention layer would do.

With the classical $N \times C \times T$ representation we can implement the products by W_Q, W_K, and W_V as convolutions.

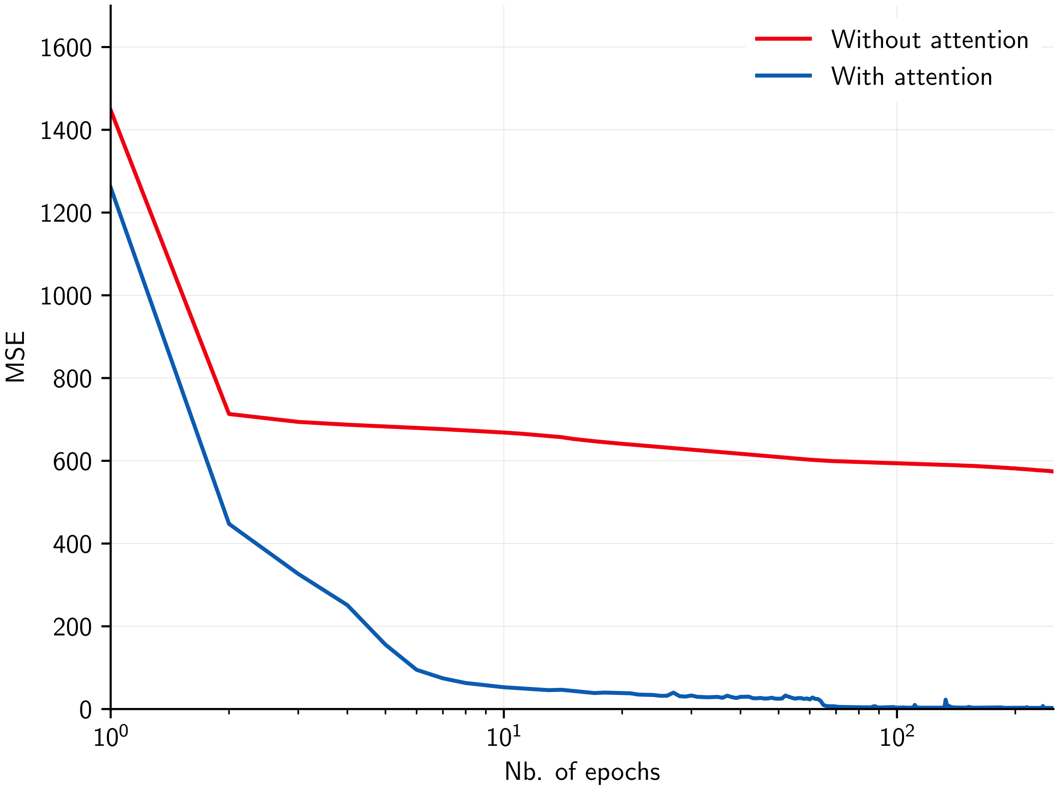
To manipulate more clearly the dimensions we use `torch.permute()` that allows to reorder them arbitrarily.

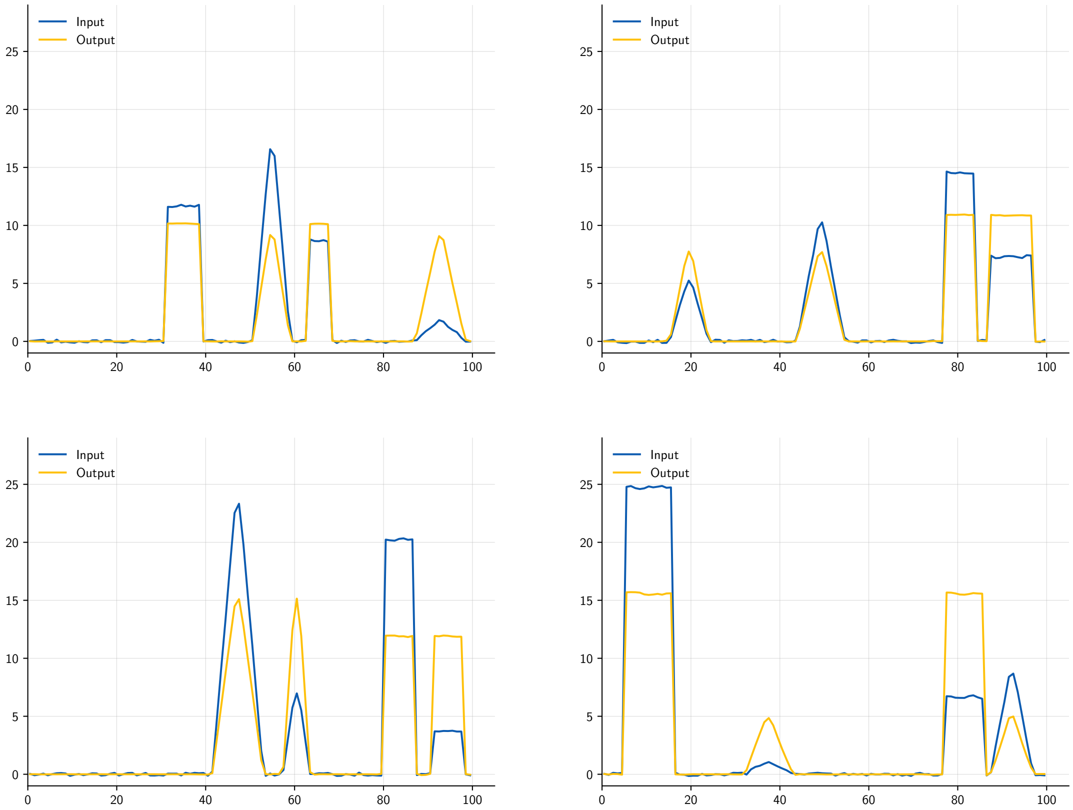
To compute $QK^\top$ and AV we need a batch matrix product, which is provided by `torch.matmul()`.

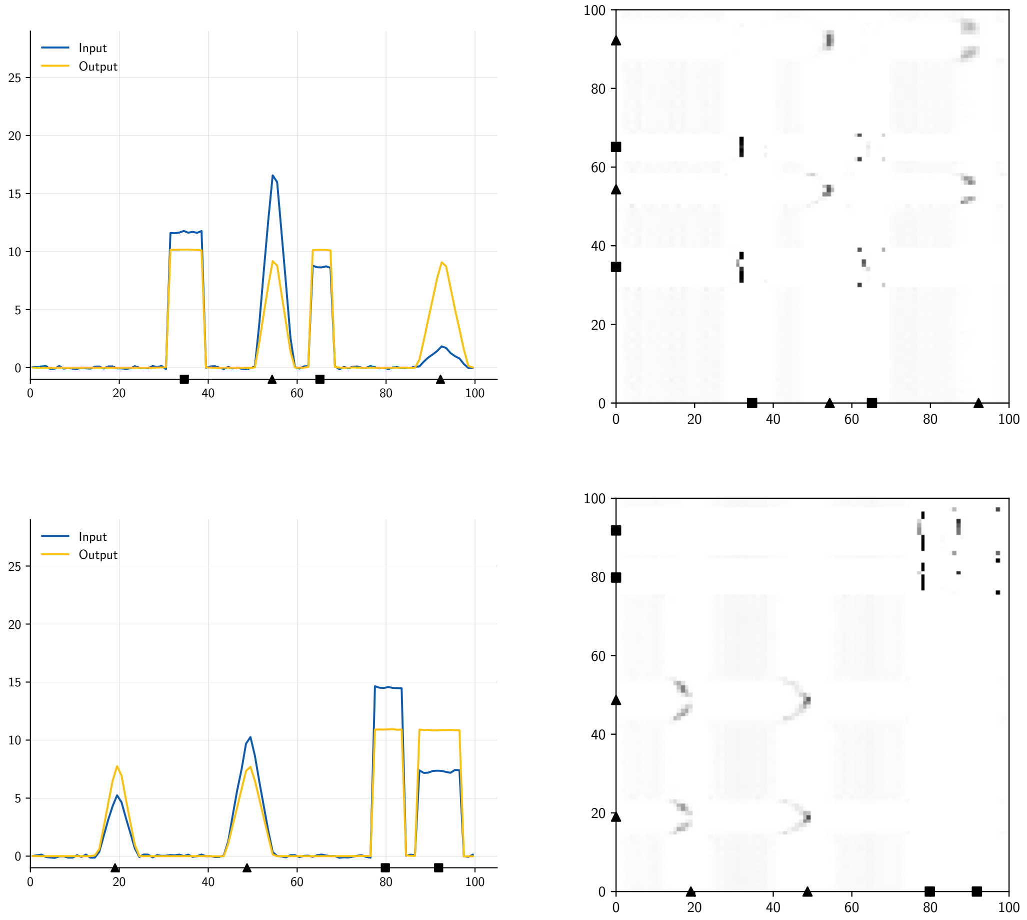
```
>>> a = torch.rand(11, 9, 2, 3)
>>> b = torch.rand(11, 9, 3, 4)
>>> m = a.matmul(b)
>>> m.size()
torch.Size([11, 9, 2, 4])
>>>
>>> m[7, 1]
tensor([[0.8839, 1.0253, 0.7473, 1.1397],
        [0.4966, 0.5515, 0.4631, 0.6616]])
>>> a[7, 1].mm(b[7, 1])
tensor([[0.8839, 1.0253, 0.7473, 1.1397],
        [0.4966, 0.5515, 0.4631, 0.6616]])
>>>
>>> m[3, 0]
tensor([[0.6906, 0.7657, 0.9310, 0.7547],
        [0.6259, 0.5570, 1.1012, 1.2319]])
>>> a[3, 0].mm(b[3, 0])
tensor([[0.6906, 0.7657, 0.9310, 0.7547],
        [0.6259, 0.5570, 1.1012, 1.2319]])
```

```
class AttentionLayer(nn.Module):
    def __init__(self, in_channels, out_channels, key_channels):
        super().__init__()
        self.conv_Q = nn.Conv1d(in_channels, key_channels, kernel_size = 1, bias = False)
        self.conv_K = nn.Conv1d(in_channels, key_channels, kernel_size = 1, bias = False)
        self.conv_V = nn.Conv1d(in_channels, out_channels, kernel_size = 1, bias = False)

    def forward(self, x):
        Q = self.conv_Q(x)
        K = self.conv_K(x)
        V = self.conv_V(x)
        A = Q.permute(0, 2, 1).matmul(K).softmax(2)
        x = A.matmul(V.permute(0, 2, 1)).permute(0, 2, 1)
        return x

    def __repr__(self):
        return self._get_name() + \
            '(in_channels={}, out_channels={}, key_channels={})'.format(
                self.conv_Q.in_channels,
                self.conv_V.out_channels,
                self.conv_K.out_channels
            )
```

```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (1): ReLU()
  (2): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (3): ReLU()
  (4): AttentionLayer(in_channels=64, out_channels=64, key_channels=64)
  (5): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (6): ReLU()
  (7): Conv1d(64, 1, kernel_size=(5,), stride=(1,), padding=(2,))
)

nb_parameters 54081
```


Without attention
With attention
MSE
1600
1400
1200
1000
800
600
400
200
0
10^0
10^1
10^2
Nb. of epochs


Input
Output
25
20
15
10
5
0
0
20
40
60
80
100



Such an attention layer disregards the absolute location of the values. Given any permutation

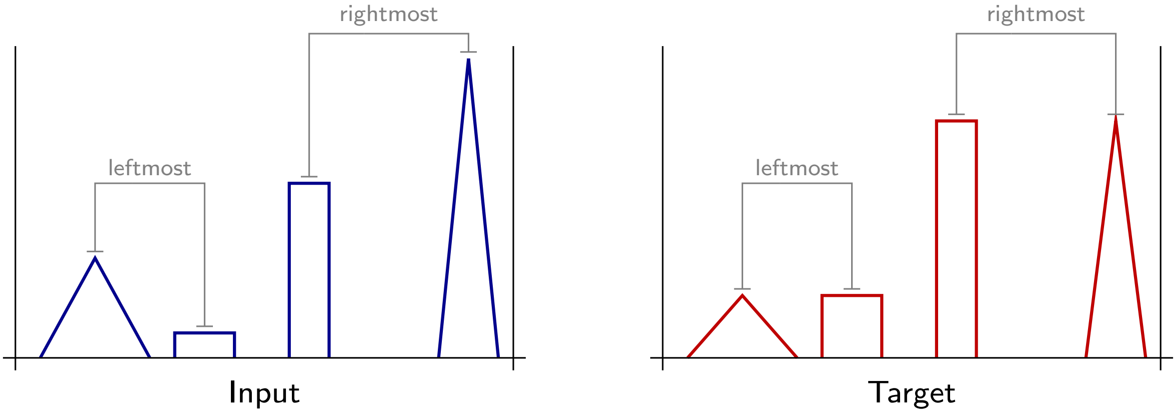
$$\sigma : \{1, \dots, S\} \to \{1, \dots, S\}$$

we have

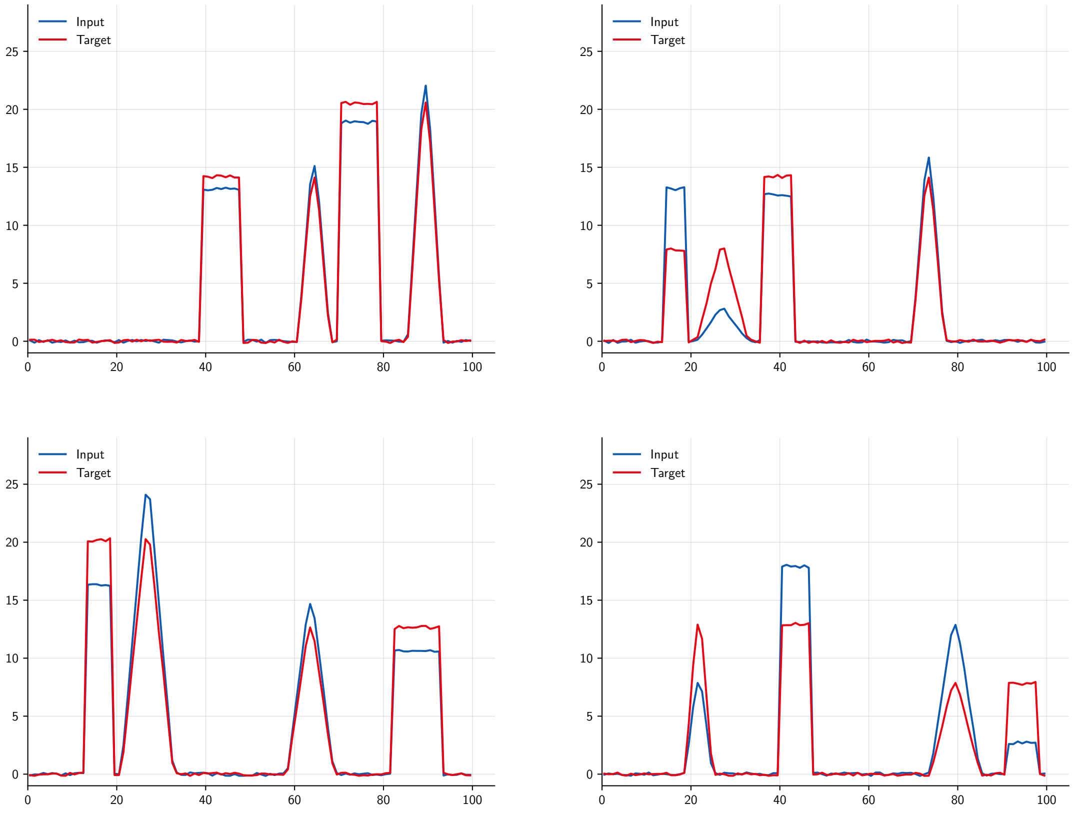
$$Y_j = \sum_i \mathrm{softmax}_i \left(Q_j K_{\sigma(i)}^\top \right) V_{\sigma(i)},$$

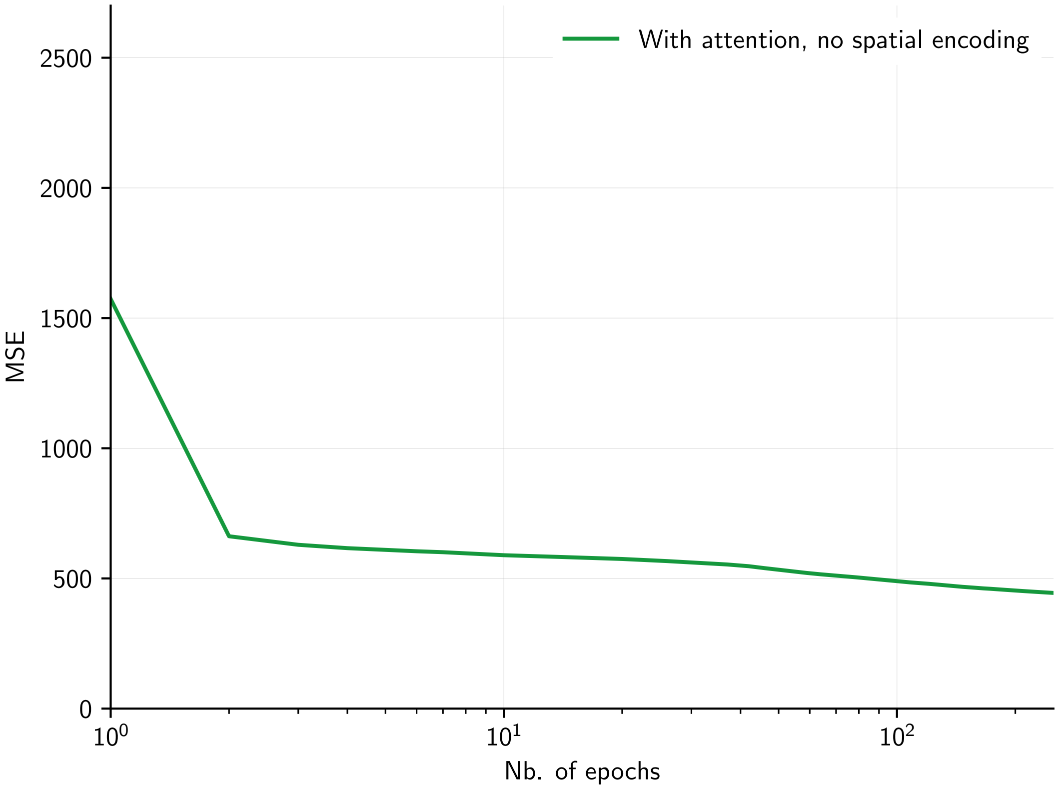
As a matter of fact, the formal definition of this operation does not requires any property on the tensor shapes. The only thing is that, except for the final dimension, Y has the same shape as Q.

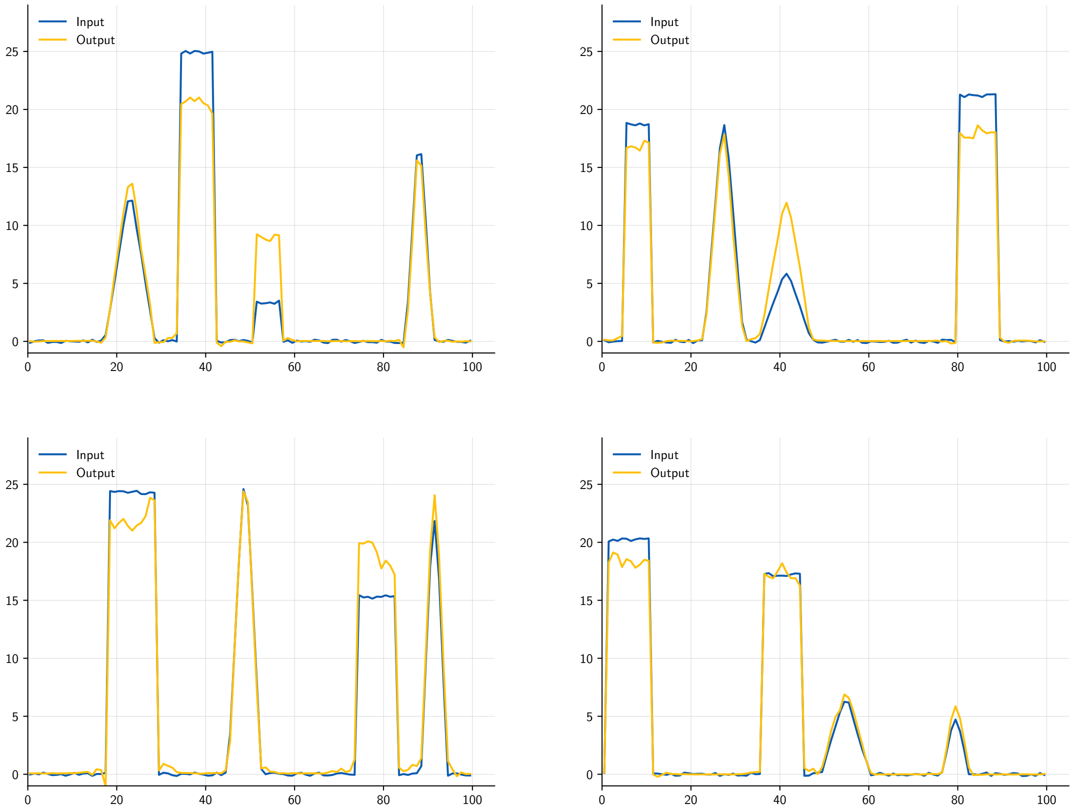
Our toy problem does not require to take into account the positioning in the tensor. We can modify it with a target where the pairs to average are the two rightmost and leftmost shapes.



Some training examples.




With attention, no spatial encoding
MSE
2500
2000
1500
1000
500
0
10^0
10^1
10^2
Nb. of epochs


Input
Output
25
20
15
10
5
0
0
20
40
60
80
100

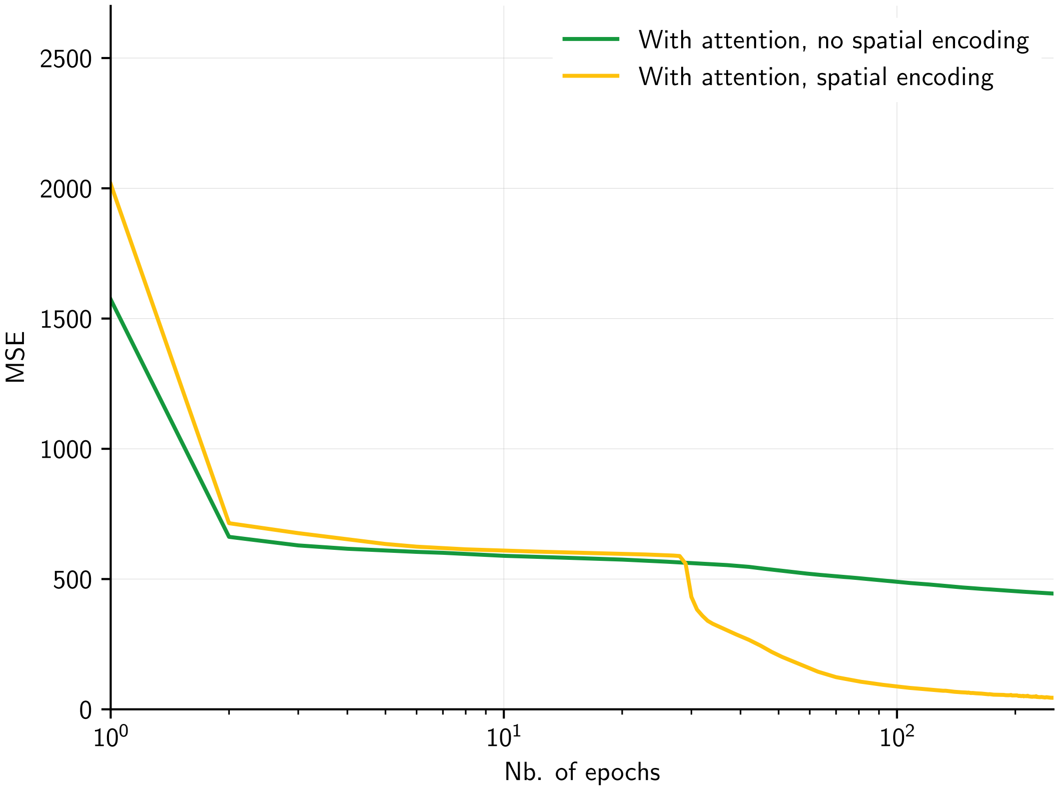
The poor performance of this model is not surprising given its inability to take into account positions in the attention layer.

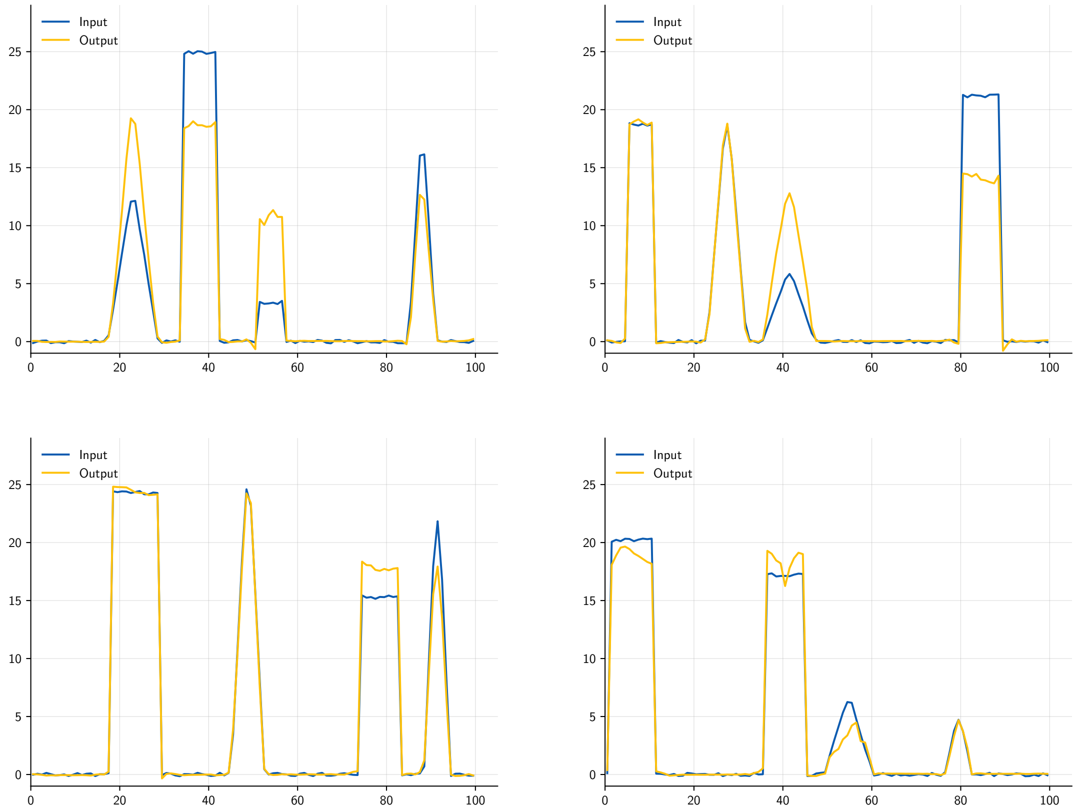
We can fix this by providing to the model a **positional encoding**.

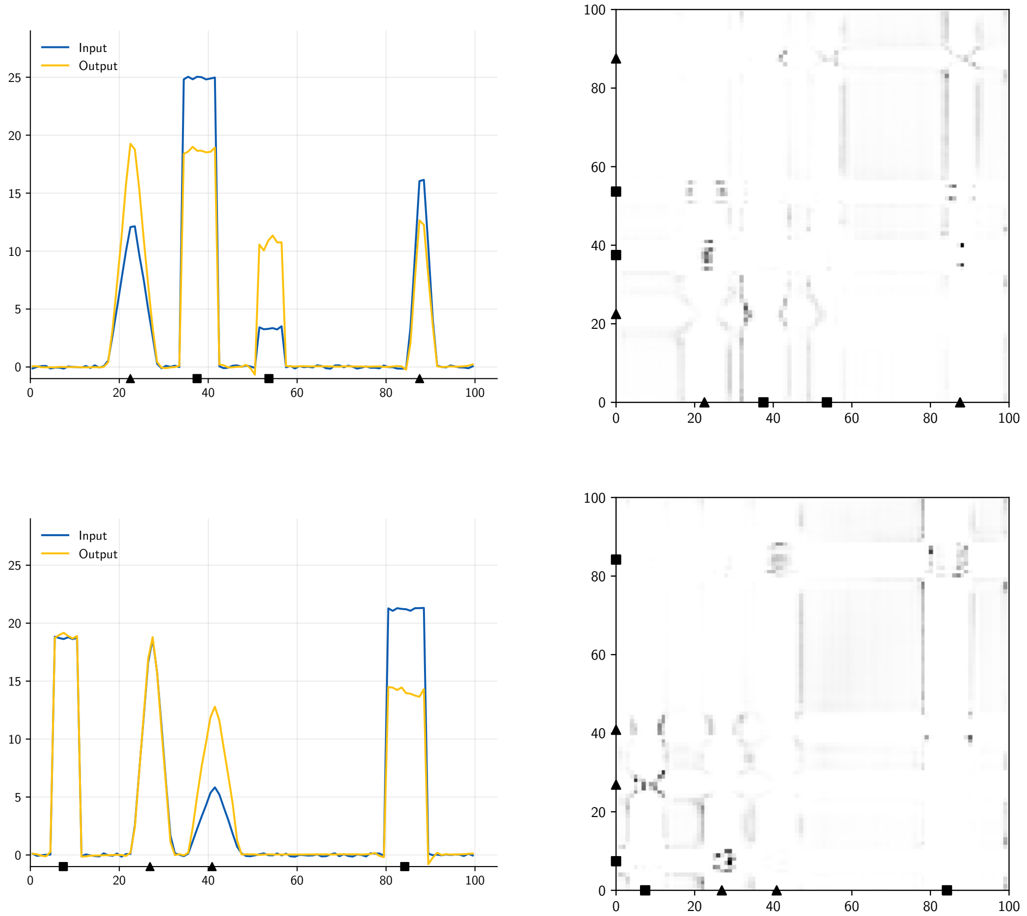
```
>>> len = 20
>>> c = math.ceil(math.log(len) / math.log(2.0))
>>> pe = (torch.arange(len)[None] // 2**torch.arange(c)[:, None])%2
>>> pe
tensor([[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
        [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]])
```

Such a tensor can simply be channel-concatenated to the input batch:

```
>>> pe = pe[None].float()
>>> input = torch.cat((input, pe.expand(input.size(0), -1, -1)), 1)
```




Input
Output
0
5
10
15
20
25
0
20
40
60
80
100


Input
Output
0
5
10
15
20
25
0
20
40
60
80
100

References

A. Graves, G. Wayne, and I. Danihelka. **Neural turing machines**. CoRR, abs/1410.5401, 2014.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. **Attention is all you need**. CoRR, abs/1706.03762, 2017.