Deep learning

13.2. Attention Mechanisms

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

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.

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.

Given a query sequence $Q \in \mathbb{R}^{T \times D}$, a key sequence $K \in \mathbb{R}^{T' \times D}$, and a value sequence $V \in \mathbb{R}^{T' \times D'}$, compute an attention matrix $A \in \mathbb{R}^{T \times T'}$ by matching Qs to Ks, and weight V with it to get the result sequence $Y \in \mathbb{R}^{T \times D'}$.

$$\forall i, A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right)$$
$$Y_i = V^\top A_i,$$

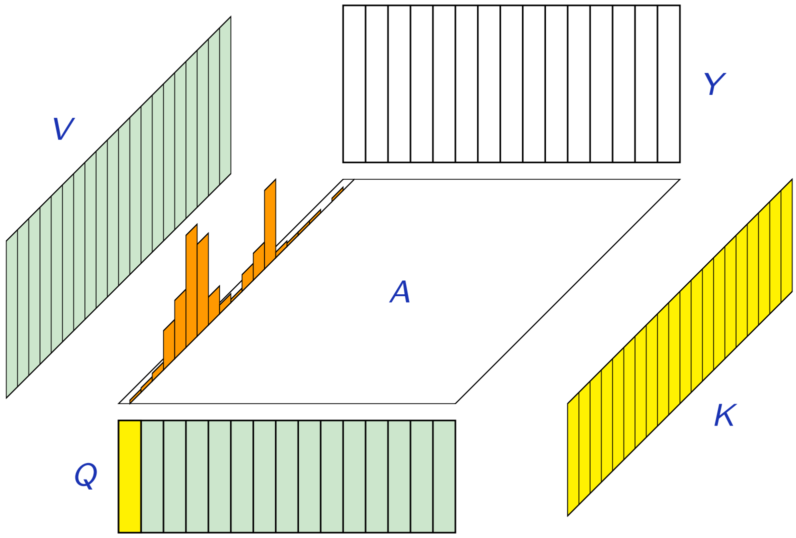
or

$$A = \text{softmax}_{row}\left(\frac{QK^\top}{\sqrt{D}}\right)$$
$$Y = AV.$$

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

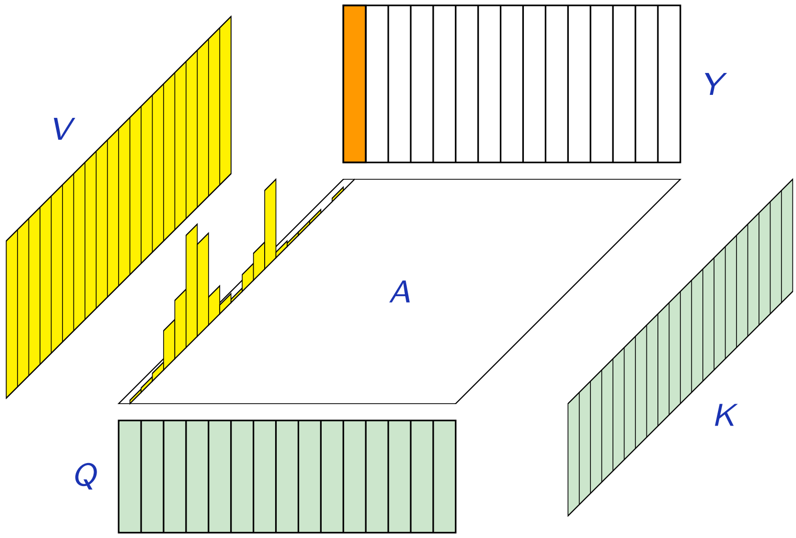
[Tensors are depicted here transposed for ease of representation.]

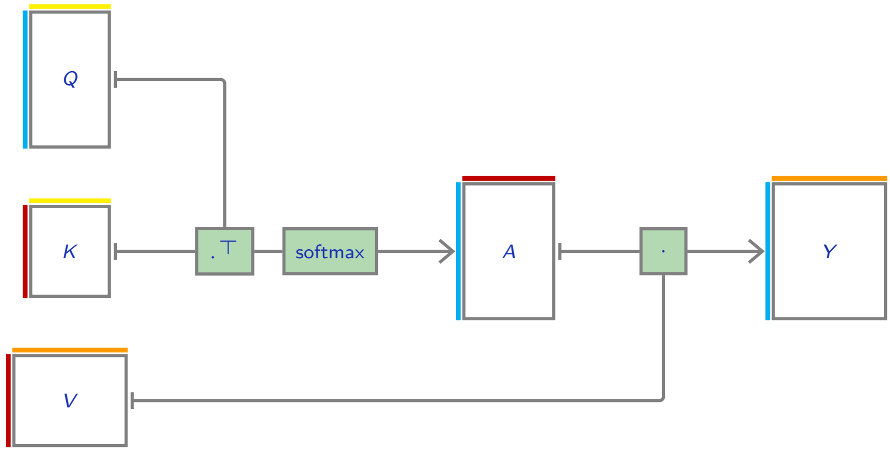
$$A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right)$$



[Tensors are depicted here transposed for ease of representation.]

$$Y_i = V^\top A_i$$



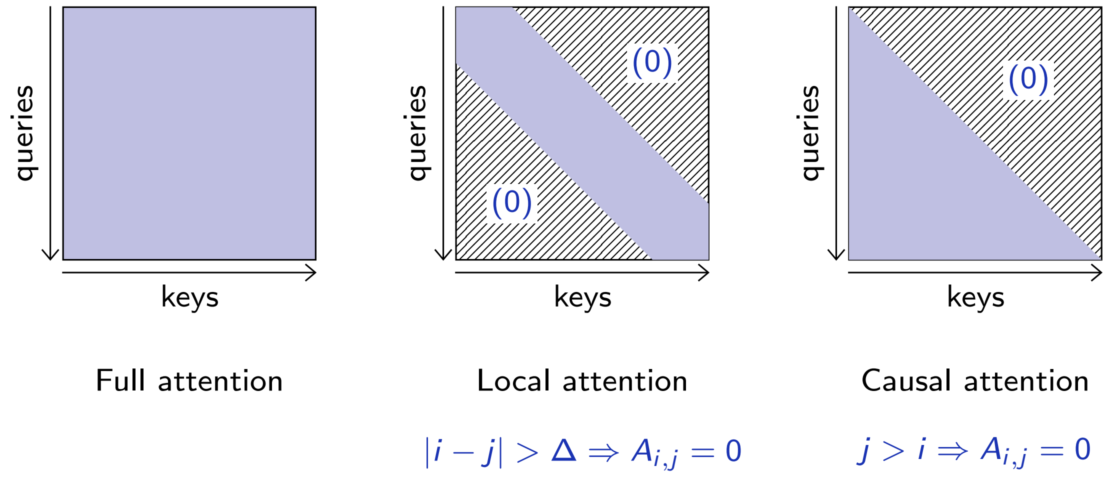


$$A = \text{softmax}_{row}\left(\frac{QK^\top}{\sqrt{D}}\right)$$

$$Y = AV.$$

Standard attention

It may be useful to mask the attention matrix, for instance in the case of self-attention, for computational reasons, or to make the model causal for auto-regression.



Notes

In the case of local attention (Beltagy et al., 2020), only keys stored near the query in the sequence are considered:

$$|i-j| > \Delta \Rightarrow A_{i,j} = 0.$$

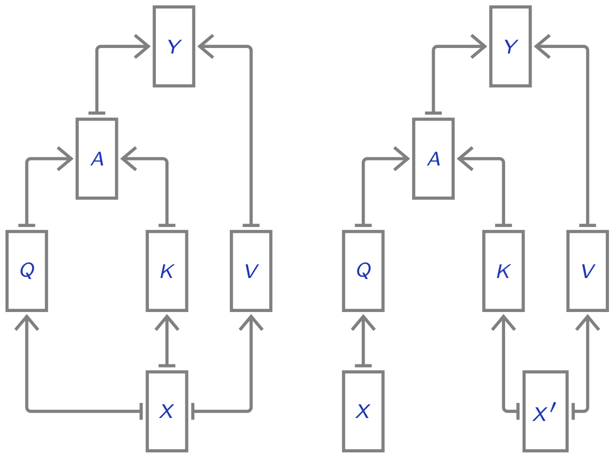
For causal attention (Vaswani et al., 2017), only keys stored before the query in the sequence are considered:

$$j > i \Rightarrow A_{i,j} = 0.$$

Attention layers

A standard attention layer takes as input two sequences X and X' and computes the tensors K, V, and Q as per-row linear functions.

$$Q = X W^{Q\top}$$
$$K = X' W^{K\top}$$
$$V = X' W^{V\top}$$
$$A = \text{softmax}_{row}\left(\frac{QK^\top}{\sqrt{D}}\right)$$
$$Y = AV$$



When $X = X'$, this is **self attention**, otherwise it is **cross attention.**

Multi-head attention combines several such operations in parallel, and Y is the concatenation of the results along the feature dimension to which is applied one more linear transformation.

Notes

The terminology of attention mechanism comes from the paradigm of key-value dictionaries for data storage in which objects (the values) are stored using a key.
Querying the database consists of matching a query with the keys of the database to retrieve the values associated to them.
This is why matrices Q and K have the same number of columns, that correspond to the dimension D of individual keys or queries because we computes matches between them. The matrices K and V have the same number of rows T' because each value is “indexed” by one key.
Each row Y_j of the output corresponds to a weighted average of the values modulated by how much the query matched the associated key.

Given a permutation σ and a $2d$ tensor X, let us use the following notation for the permutation of the rows: $\sigma(X)_i = X_{\sigma(i)}$.

The standard attention operation is **invariant to a permutation of the keys and values:**

$$Y(Q, \sigma(K), \sigma(V)) = Y(Q, K, V),$$

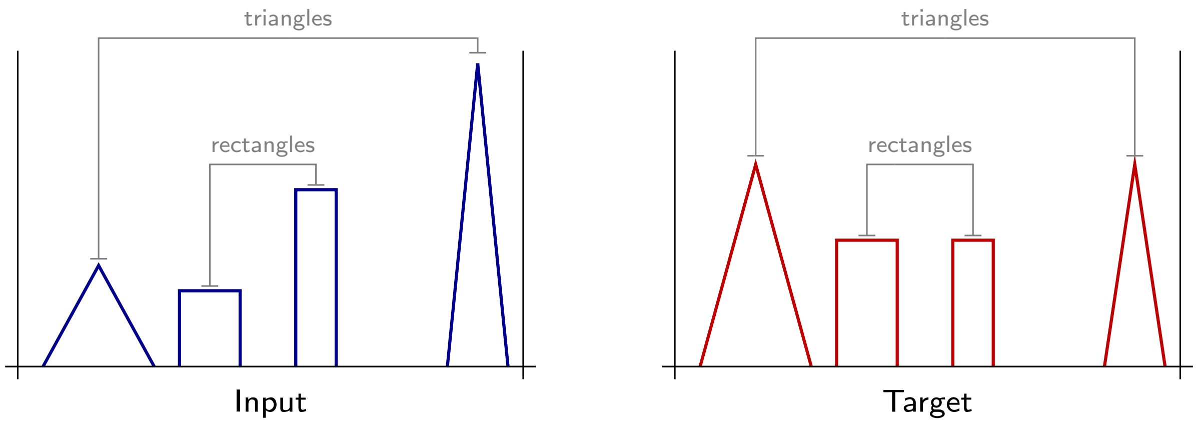
and **equivariant to a permutation of the queries**, that is the resulting tensor is permuted similarly:

$$Y(\sigma(Q), K, V) = \sigma(Y(Q, K, V)).$$

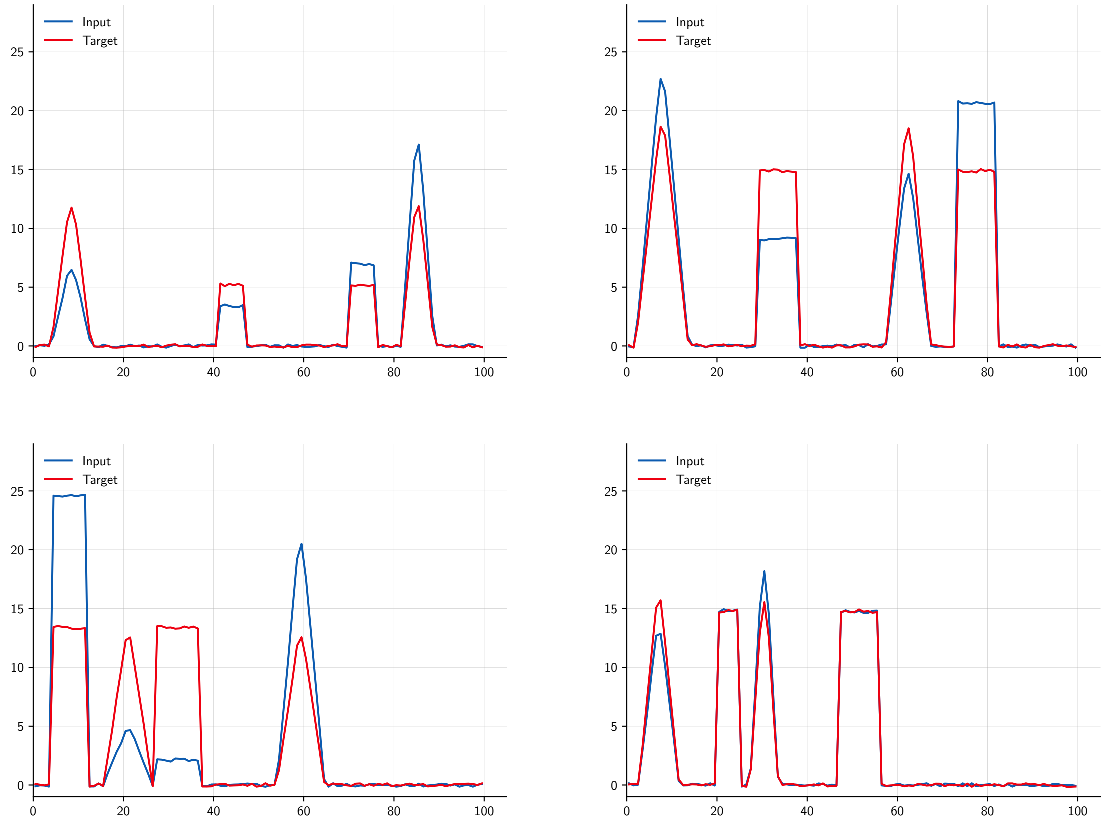
Consequently self attention and cross attention are equivariant to permutations of X, and cross attention is invariant to permutations of X'.

To illustrate the behavior of such an attention layer, we consider a toy sequence-to-sequence problem with sequences composed of two triangular and two rectangular patterns.

The target averages the heights in each **pair of shapes**.



Some training examples.



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

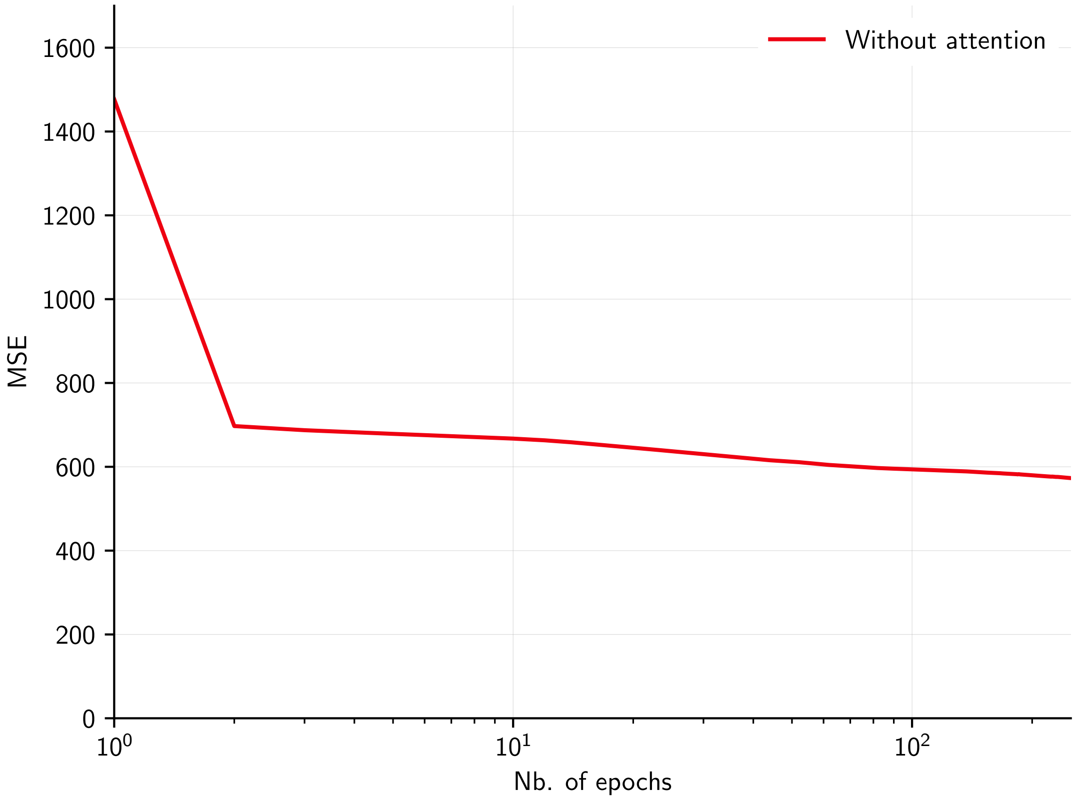
```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (1): ReLU()
  (2): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (3): ReLU()
  (4): Conv1d(64, 64, kernel_size=(5,), stride=(1,), padding=(2,))
  (5): ReLU()
  (6): Conv1d(64, 64, kernel_size=(5,), stride=(1,),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

nb_parameters 62337
```


Notes

As a baseline, we consider a simple convolutional network which takes as input the 1d sequence, procss them with four hidden layers with 64 channels, and outputs a new 1d sequence.
Adequate padding preserves the length of the sequence.

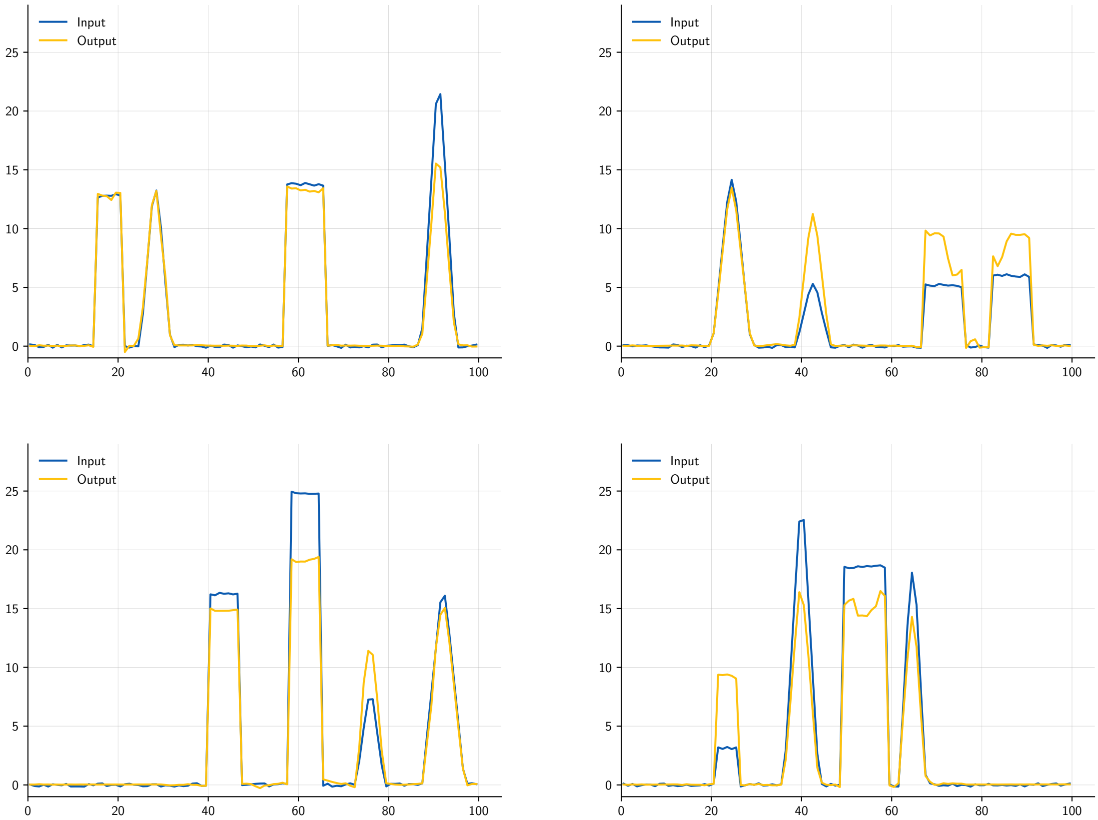
Training is done with the MSE loss and Adam.

```
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()
```



Notes

With such a simple model and no attention, the loss remains high. One epoch consists of 25,000 samples.



Notes

These are example of test results, showing the input in blue, and the generated sequences in yellow.
The output is not that great, which is consistent with the training loss remaining high, although we can notice that the model sometimes pushes towards the mean when the elements of a pair are close.

The poor performance of this model is not surprising given its inability to transport 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.

We implement our own self attention layer with tensors $N \times C \times T$ so that the products by W_Q, W_K, and W_V can be implemented as convolutions.

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]])
```


Notes

`a` can be interpreted as a 11×9 matrix of 2×3 matrices, and `b` as a 11×9 matrix of 3×4 matrices.
`matmul` loops over the first dimensions 11×9 to perform every time the product between the matrices of size 2×3 and 3×4.
The overall operation results in a 11×9 matrix of 2×4 matrices.

```
class SelfAttentionLayer(nn.Module):
    def __init__(self, in_dim, out_dim, key_dim):
        super().__init__()
        self.conv_Q = nn.Conv1d(in_dim, key_dim, kernel_size = 1, bias = False)
        self.conv_K = nn.Conv1d(in_dim, key_dim, kernel_size = 1, bias = False)
        self.conv_V = nn.Conv1d(in_dim, out_dim, 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.transpose(1, 2).matmul(K).softmax(2)
        y = A.matmul(V.transpose(1, 2)).transpose(1, 2)
        return y
```

Note that for simplicity it is single-head attention, and the $1/\sqrt{D}$ is missing.

The computation of the attention matrix A and the layer's output Y could also be expressed somehow more clearly with Einstein summations (see lecture 1.5. "High dimension tensors") as

```
A = torch.einsum('nct,ncs->nts', Q, K).softmax(2)
y = torch.einsum('nts,ncs->nct', A, V)
```


Notes

To link between the notations introduced earlier and the current implementation, we have:

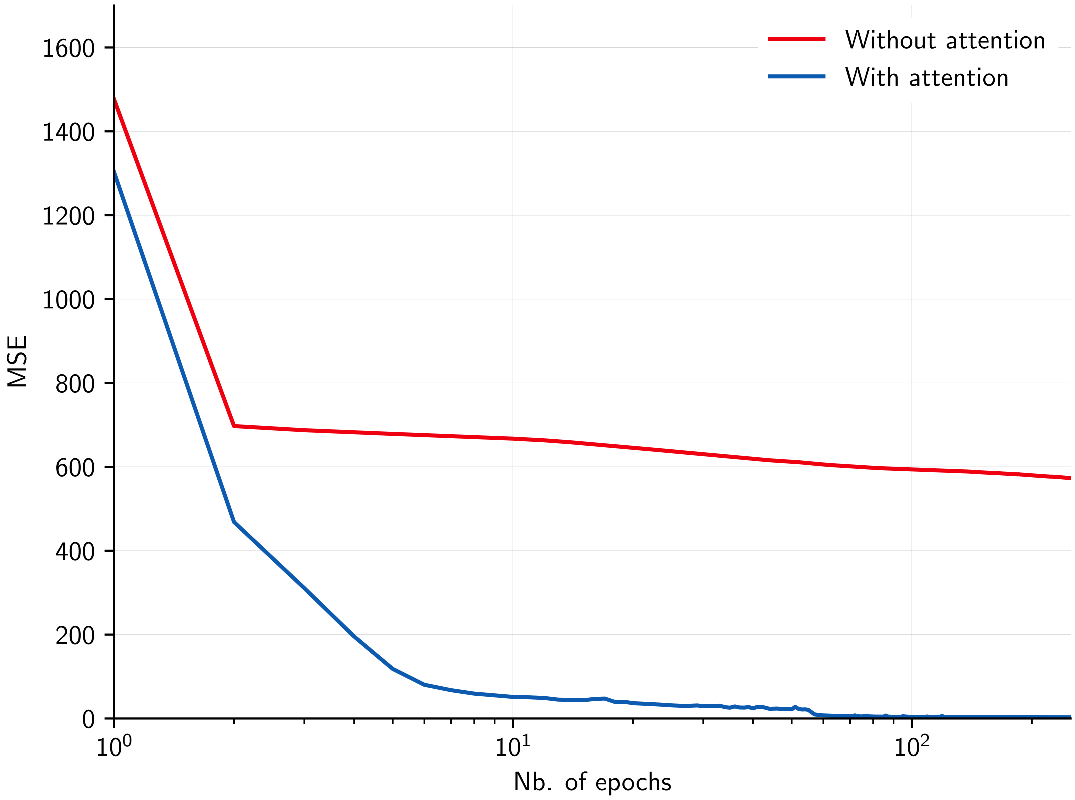
- $X = X'$ for self attention,
- $T = T'$ since the self attention has as many queries as values,
- $D =$ `key_dim`, and
- $D' =$ `out_dim`.

The forward function takes as input a batch of size $N \times C \times T$, so that the products by W_Q, W_K, and W_V are implemented with 1d convolutions. Since the channel comes first per sample, to compute the attention matrix $A = QK^\top$, we transpose [the two last dimensions of] Q. And similarly to compute AV, we need to transpose [the two last dimensions of] V.

```
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): SelfAttentionLayer(in_dim=64, out_dim=64, key_dim=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
```

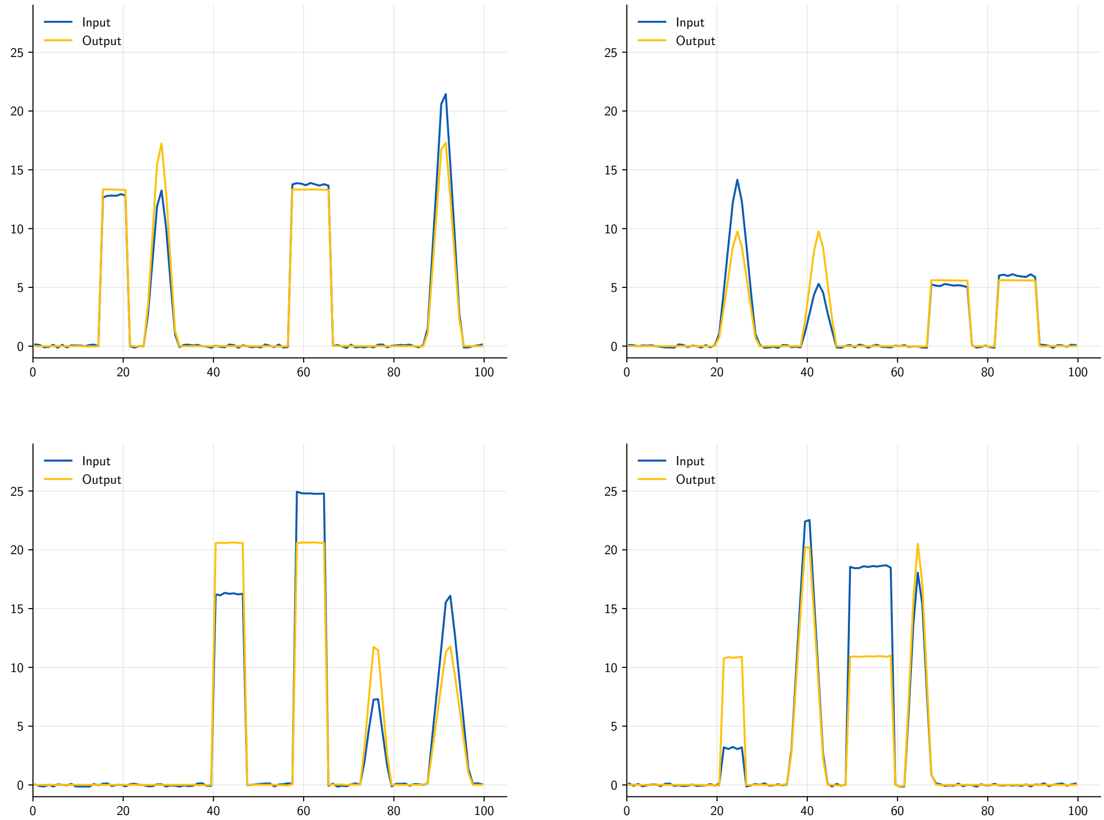

Notes

We modify our convolutional baseline by replacing the middle convolution layer and the following ReLU with the attention layer we have implemented. We choose for the key dimension the same as for the values, that is the number of channels.
Note that the resulting number of parameters is slightly less that with the previous convolutional network.



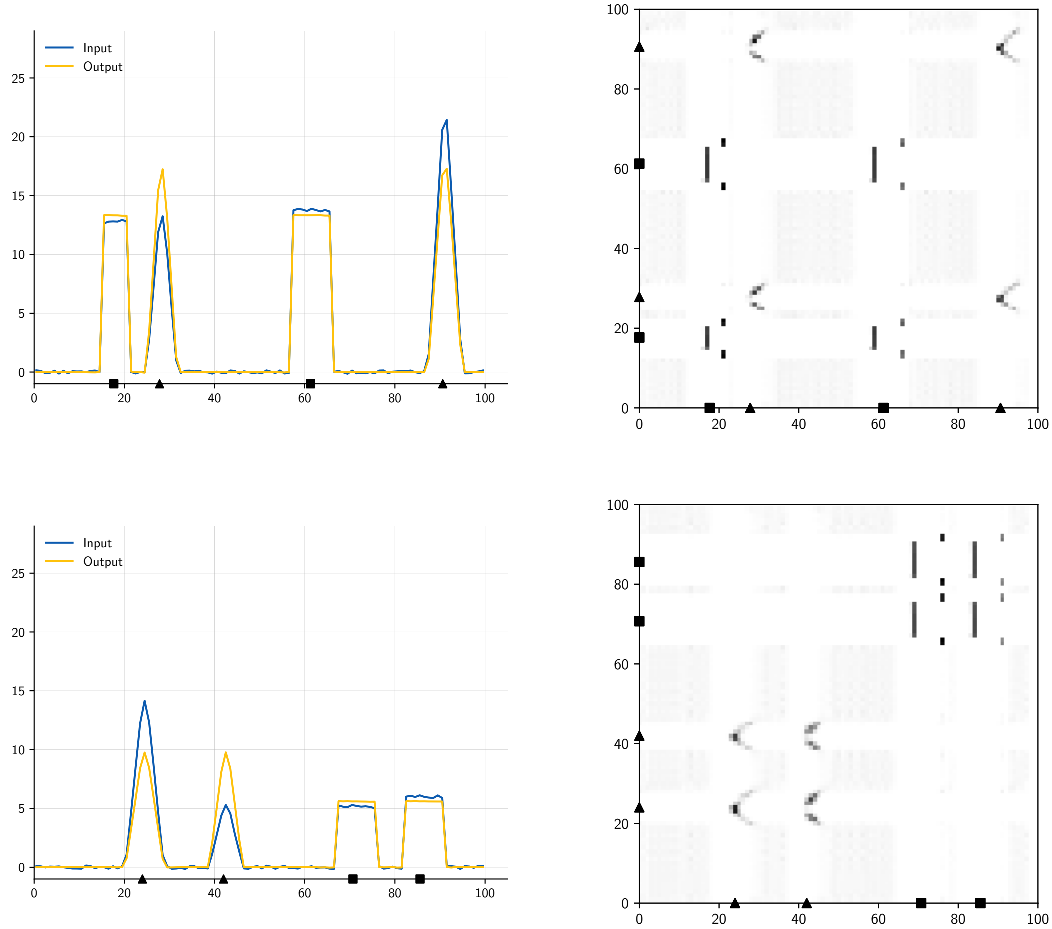
Notes

The exact same training procedure yields much better results with the attention layer, as the loss goes down to zero.



Notes

These are example results obtained with the attention network, showing the input in blue, and the generated sequences in yellow.
The network does what is it supposed to do. We can see that the height of each pair is now averaged properly.

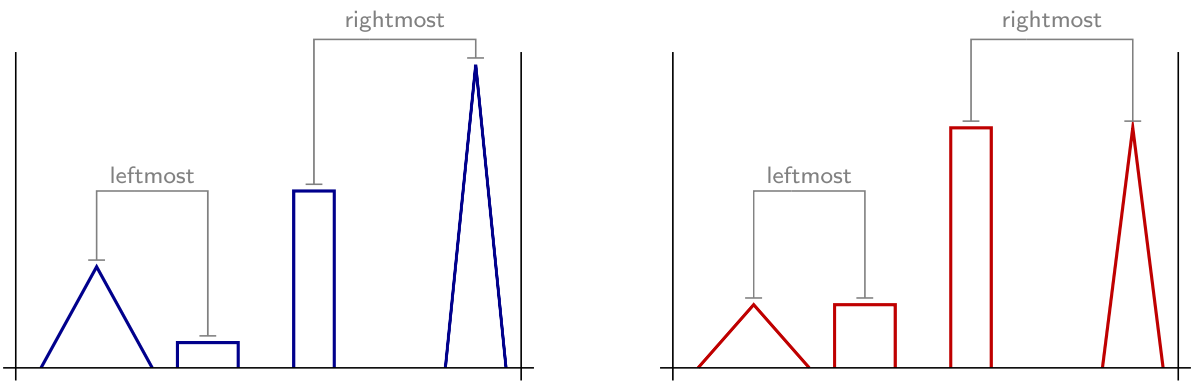


Notes

The images on the left are test sequences. Markers are placed at the indexes of the sequence corresponding to the shape centers: black squares for the rectangles, and black triangles for triangles.
The images on the right are the attention matrices, with white standing for small coefficients and black for large ones.
This shows that each pair of shapes attend at each other. Rectangles put attention on the boundary of their edges, and the triangles put emphasis on their respective slopes.

Because it is invariant to a permutation of the keys and values, such an attention layer disregards the absolute location of the values.

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.

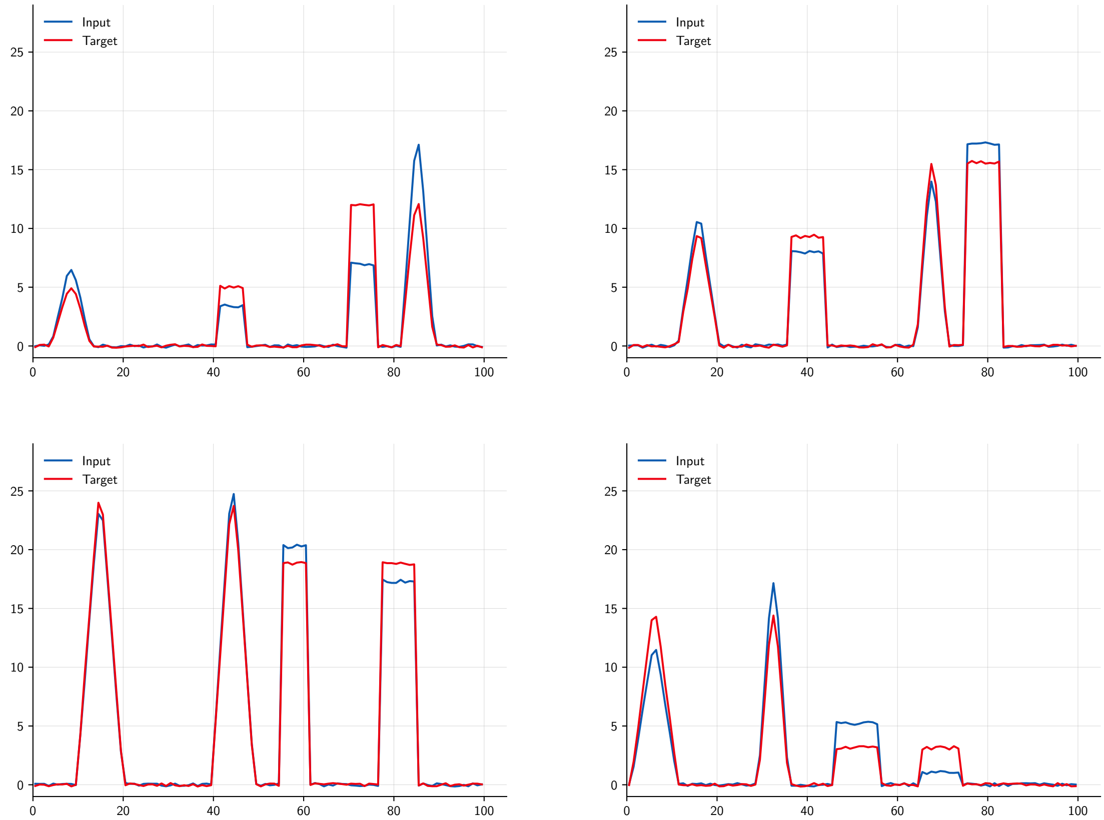


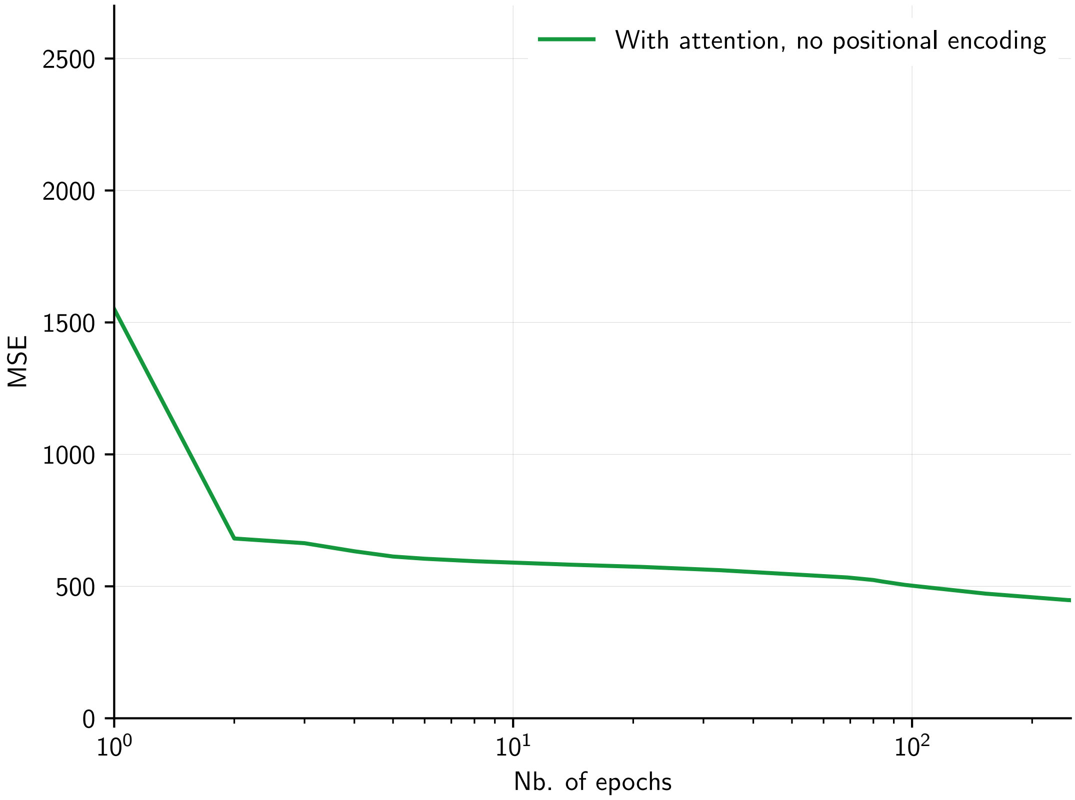
Input Target

Notes

To illustrate this drawback, we design a new synthetic task in which the goal is to average the heights of the two leftmost shapes with each other, and the heights of the two rightmost with each other.
Such a task still requires attention, because it involves looking at features far away from one another, but be able to take into account locations.

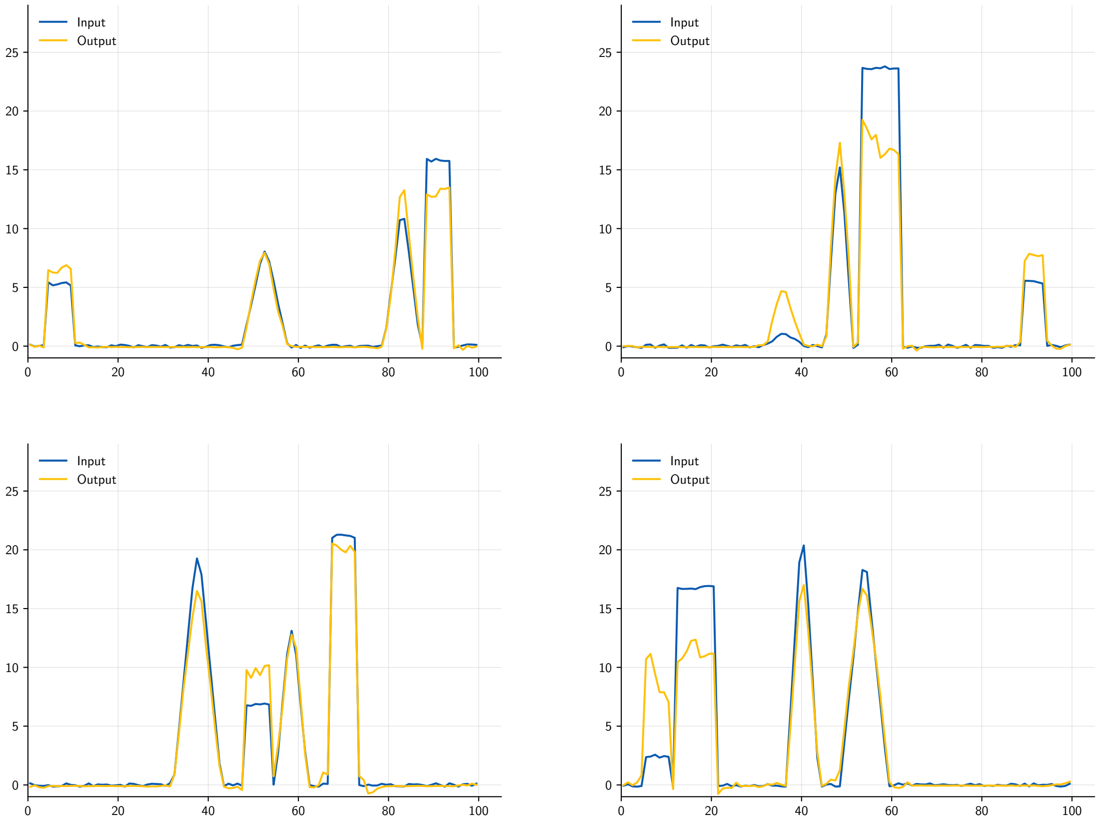
Some training examples.





Notes

Our attention model on this new task performs almost as badly as the convolutional network on the first task.



Input
Output
0
5
10
15
20
25
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**.

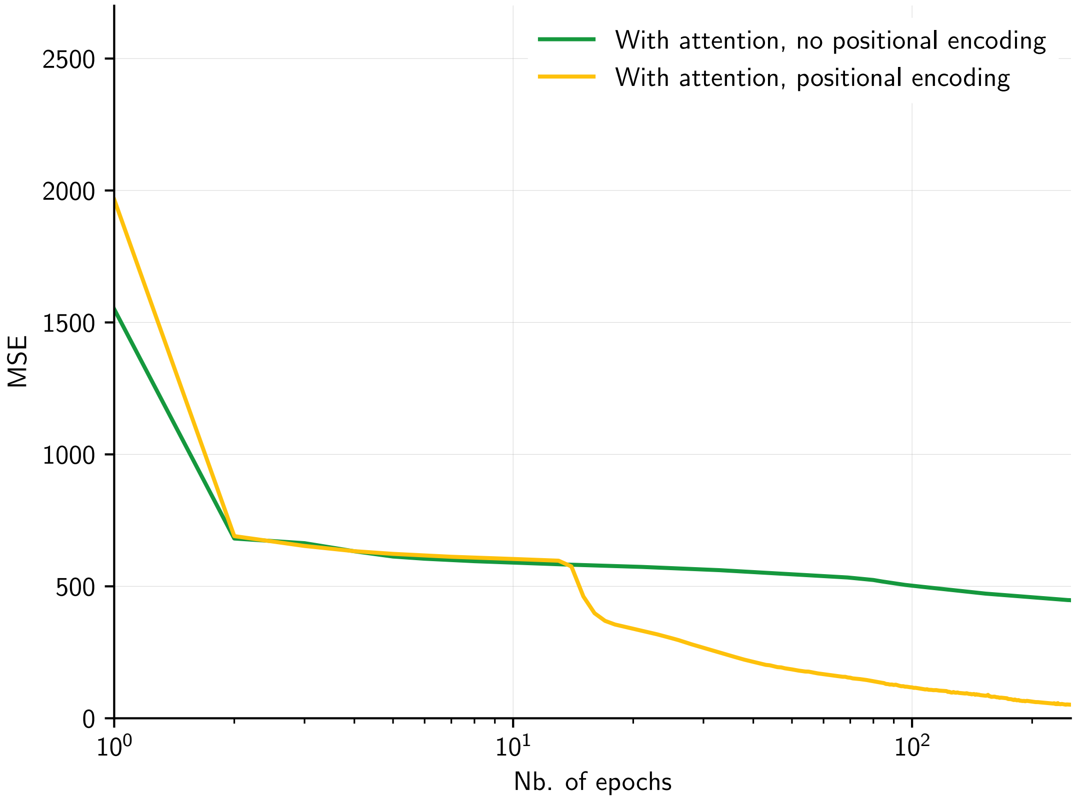
```
>>> len = 20
>>> c = math.ceil(math.log(len) / math.log(2.0))
>>> o = 2**torch.arange(c).unsqueeze(1)
>>> pe = (torch.arange(len).unsqueeze(0).div(o, rounding_mode = 'floor')) % 2
>>> pe
tensor([[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 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)
```

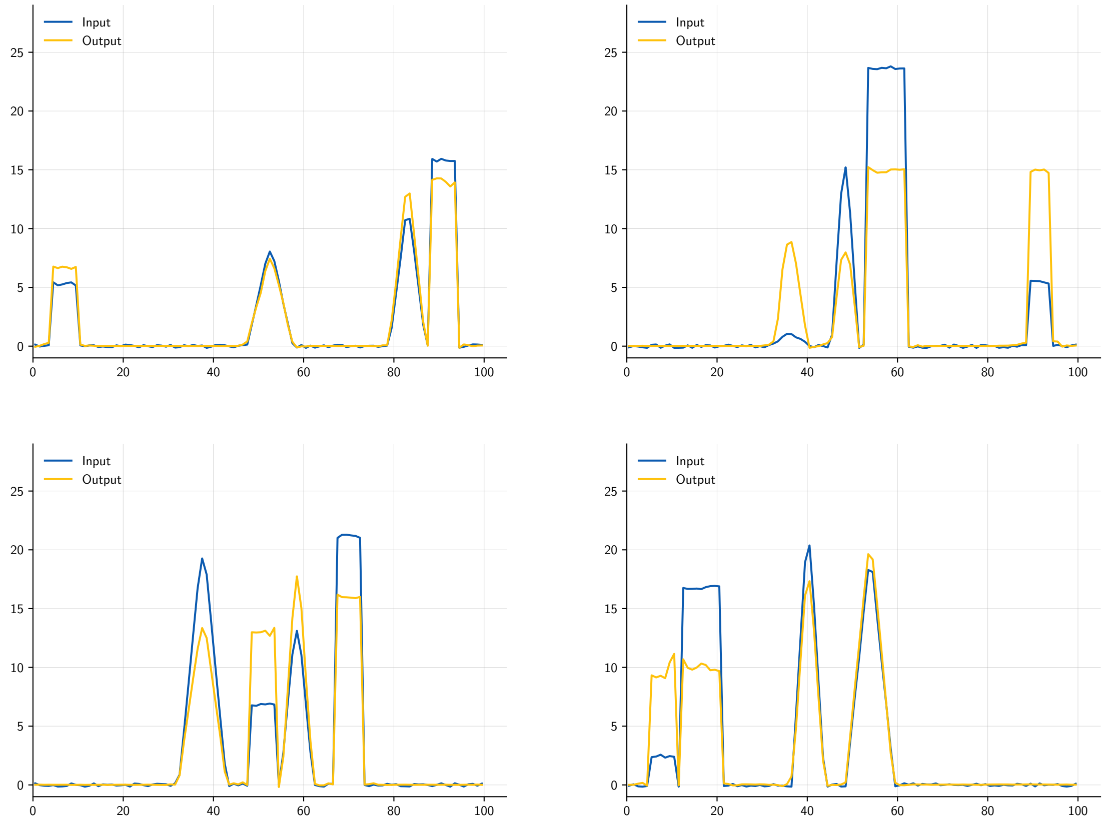

Notes

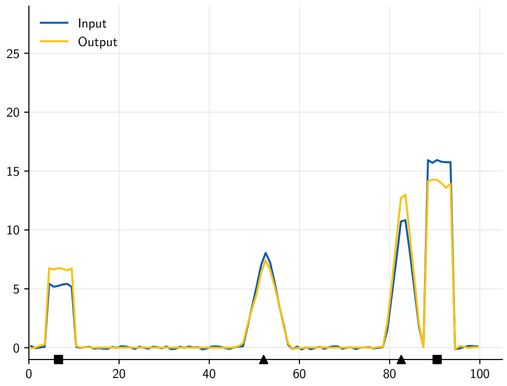
The positional encoding aims at augmenting the input tensor with a binary code which completely determines the location in the sequence. With a sequence of length 20, $B = 5$ channels suffice: the first element is associated to code $(0, 0, 0, 0, 0)$, the second to $(0, 0, 0, 0, 1)$, etc. which are the binary encoding of the index.
A minibatch of N samples representing sequences of T elements of dimension D, is of size $N \times D \times T$. After the positional encoding is concatenated as channels to the dimension of the elements, the minibatch is of shape $N \times (D + B) \times T$.
Other coding scheme exists, for instance using trigonometric functions instead of a hard binary encoding.

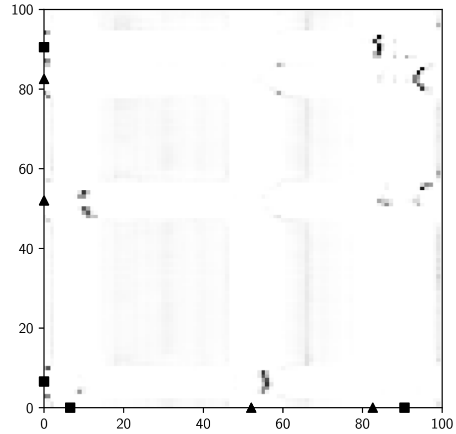


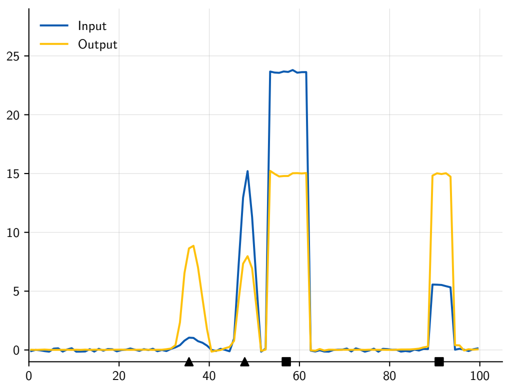
Notes

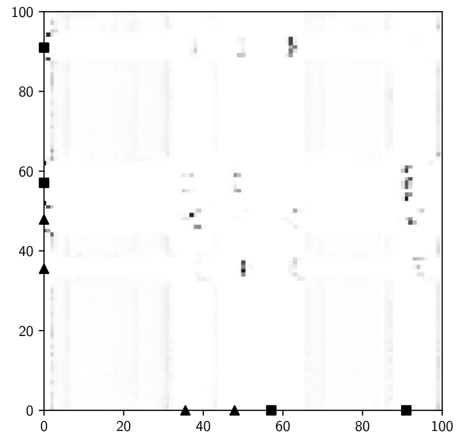
The graph shows the training losses of our attention model with and without positional encoding.


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









Notes

The images on the left are test sequences. Markers are placed at the indexes of the sequence corresponding to the shape centers: black squares for the rectangles, and black triangles for triangles.
The images on the right are the attention matrices, with white standing for small coefficients and black for large ones.
Although not as strong as in the previous task, we can see that the attention is put on the first two shapes jointly, and on the last two jointly.

References

I. Beltagy, M. Peters, and A. Cohan. **Longformer: The long-document transformer**. CoRR, abs/2004.05150, 2020.

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.