Deep learning

13.2. Attention Mechanisms

François Fleuret
https://fleuret.org/dlc/
The simplest form of attention is **content-based attention**. Given an “attention function”

\[ a : \mathbb{R}^D \times \mathbb{R}^C \rightarrow \mathbb{R} \]

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

\[
\forall j = 1, \ldots, T, \quad Y_j = \sum_{i=1}^{S} \frac{\exp(a(V_i; \theta_j))}{\sum_{k=1}^{T} \exp(a(V_k; \theta_j))} V_i \\
= \sum_{i=1}^{S} \text{softmax}_i \left( a(V_i; \theta_j) \right) V_i.
\]

---

**Notes**

The attention function takes as input two vectors, and outputs an attention score. \( \theta \) is a parameter of the model which is optimized during training.

Each row of the output \( Y \) is a weighted average of the rows of \( V_i \) with weights that depend on the attention. For a computed component \( Y_j \), the weights of the values \( V_i \) are modulated by the by \( \theta_j \) which specifies what this component \( j \) attends to.
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
\[ \forall j = 1, \ldots, T, \ Y_j = \sum_{i=1}^{S} \text{softmax}_i \left( a \left( C_j, V_i; \theta \right) \right) V_i. \]

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

$$Y = \text{softmax} \left( Q K^\top \right) V.$$

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.

---

**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.
Notes

This figure depicts the computation made by an attention mechanism. Each array represents a matrix and the intensity of the pixels correspond to the values of the coefficients: blue for negative, white for zero and red for positive.

The matrix $QK^\top$ can be interpreted as the queries / keys matching scores.

E.g. the first row of $K$ (first column of $K^\top$) consists of coefficients $\simeq 0$ everywhere but the last one largely positive. This key matches the 5th, 7th, and 9th rows of $Q$, which have all the same structure. This results in a large matching coefficients in the 1st column of the 5th, 7th, and 9th rows of $QK^\top$.

The matrix $A$ is obtained from $QK^\top$ with a softmax in each row, hence has only positive entries, and keeps the same locations for the high / low matching scores in each row.

Since the 5th, 7th, and 9th rows of the query match strongly only the first row of the keys, the attention for the resulting rows is concentrated on the 3rd row of $V$, resulting in the corresponding rows of $Y$ to be almost perfect copies of that row from $V$. 

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

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

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.
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, processes 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.

```python
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 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 compute $QK^T$ 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 \(11 \times 9\) matrix of \(2 \times 3\) matrices, and \(b\) as a \(11 \times 9\) matrix of \(3 \times 4\) matrices.

\texttt{matmul} loops over the first dimensions \(11 \times 9\) to perform every time the product between the matrices of size \(2 \times 3\) and \(3 \times 4\).

The overall operation results in a \(11 \times 9\) matrix of \(2 \times 4\) matrices.
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.transpose(1, 2).matmul(K).softmax(2)
        y = A.matmul(V.transpose(1, 2)).transpose(1, 2)
        return y

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”)

\[
A = \text{torch.einsum('nct,ncs->nts', Q, K).softmax(2)}
\]
\[
y = \text{torch.einsum('nts,ncs->nct', A, V)}
\]

Notes

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

- $X = X'$, as self-attention from slide 6,
- $T = T'$ since the self attention has as many queries as values,
- $D = \text{key_channels}$, and
- $D' = \text{out_channels}$.

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 $A = QK^T$, 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): 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

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 than 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 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.
Such an attention layer disregards the absolute location of the values. Given any permutation
\[ \sigma : \{1, \ldots, S\} \to \{1, \ldots, S\} \]
we have
\[ Y_j = \sum_i \text{softmax}_i \left( Q_j K_{\sigma(i)}^\top \right) V_{\sigma(i)}, \]

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.

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.
Francois Fleuret

Deep learning / 13.2. Attention Mechanisms

25 / 29
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.

```python
>>> 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, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]])
```

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

```python
>>> 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.
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
