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”

\[
a : \mathbb{R}^{D'} \times \mathbb{R}^{D} \to \mathbb{R}
\]

and model parameters

\[
\theta \in \mathbb{R}^{T \times D}
\]

this operation takes a “value” tensor as input

\[
V \in \mathbb{R}^{T' \times D'}
\]

and computes an output

\[
Y \in \mathbb{R}^{T \times D'}
\]

with

\[
\forall j = 1, \ldots, T, \quad Y_j = \frac{\sum_{i=1}^{T'} \exp(a(V_i; \theta_j))}{\sum_{k=1}^{T} \exp(a(V_k; \theta_j))} V_i = \sum_{i=1}^{T'} \text{softmax}_i (a(V_i; \theta_j)) V_i.
\]
This differs from context attention, which, given two inputs: a “context” tensor

\[ C \in \mathbb{R}^{T \times D} \]

and a “value” tensor

\[ V \in \mathbb{R}^{T' \times D'} \]

computes a tensor

\[ Y \in \mathbb{R}^{T \times D'} \]

with

\[ \forall j = 1, \ldots, T, \ Y_j = \sum_{i=1}^{T'} \text{softmax}_i (a (C_j, V_i; \theta)) \ V_i. \]
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 \textbf{values} associated to \textbf{keys} matching a \textbf{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 $Q$s to $K$s, 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

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

$$Y = AV$$

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

\[ A_i = \text{softmax} \left( \frac{KQ_i}{\sqrt{D}} \right) \]
Tensors are depicted transposed here for ease of representation.

\[ Y_i = V^\top A_i \]
Tensors are depicted transposed here for ease of representation.

\[ A_i = \text{softmax} \left( \frac{KQ_i}{\sqrt{D}} \right) \]
Tensors are depicted transposed here for ease of representation.

\[ Y_i = V^\top A_i \]
Tensors are depicted transposed here for ease of representation.

\[ A_i = \text{softmax} \left( \frac{KQ_i}{\sqrt{D}} \right) \quad Y_i = V^\top A_i \]
A standard attention layer takes as input two sequences \( X \) and \( X' \) and computes the tensors \( K, V, \) and \( Q \) as linear functions.

\[
K = W^K X \\
V = W^V X \\
Q = w^Q X' \\
Y = \text{softmax}_{row} \left( \frac{QK^T}{\sqrt{D}} \right) V
\]

When \( X = X' \), this is self-attention, otherwise cross attention.

Several such operations can be combined, where \( Y \) is the concatenation of the separate results along the feature dimension. This is multi-head attention.
A standard attention layer takes as input two sequences $X$ and $X'$ and computes the tensors $K$, $V$, and $Q$ as linear functions.

\[
K = W^K X \\
V = W^V X \\
Q = w^Q X'
\]

\[
Y = \text{softmax}_{\text{row}} \left( \frac{QK^T}{\sqrt{D}} \right) V
\]

When $X = X'$, this is **self attention**, otherwise **cross attention**.
A standard attention layer takes as input two sequences $X$ and $X'$ and computes the tensors $K$, $V$, and $Q$ as linear functions.

\[
\begin{align*}
K &= W^K X \\
V &= W^V X \\
Q &= w^Q X'
\end{align*}
\]

\[
Y = \text{softmax}_{\text{row}} \left( \frac{QK^T}{\sqrt{D}} \right) V
\]

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

Several such operations can be combined, where $Y$ is the concatenation of the separate results along the feature dimension. This is multi-head 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.
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.
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.
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.
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 \textit{pair of shapes}. 

\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
\flushleft
\textbf{Input}
\end{minipage}\hfill
\begin{minipage}{0.05}\textwidth\end{minipage}\hfill
\begin{minipage}{0.45\textwidth}
\flushright
\textbf{Target}
\end{minipage}
\end{figure}
Some training examples.
We test first a 1d convolutional network, with no attention mechanism.

```python
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.

\[
\text{batch\_size} = 100
\]

\[
\text{optimizer} = \text{torch.optim.Adam(model.parameters(), lr = 1e-3)}
\]

\[
\text{mse\_loss} = \text{nn.MSELoss()}
\]

\[
\mu, \text{std} = \text{train\_input.mean()}, \text{train\_input.std()}
\]

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) / \text{std})
        loss = \text{mse\_loss}(output, targets)

        optimizer.zero\_grad()
        loss.backward()
        optimizer.step()
Graph showing the Mean Squared Error (MSE) over the number of epochs without attention. The MSE decreases sharply at the beginning and then gradually as the number of epochs increases.
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.
We implement our own 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()`.
```python
>>> 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.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.
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

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”)

$$A = \text{torch.einsum}('nct,ncs->nts', Q, K).\text{softmax}(2)$$
$$y = \text{torch.einsum}('nts,ncs->nct', A, V)$$
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
Francois Fleuret  
Deep learning / 13.2. Attention Mechanisms
Such an attention layer disregards the absolute location of the values.

Given any permutation

$$\sigma : \{1, \ldots, S\} \rightarrow \{1, \ldots, S\},$$

we have

$$Y_j = \sum_i \text{softmax}_i \left( Q_j K_{\sigma(i)}^T \right) V_{\sigma(i)}.$$
Such an attention layer disregards the absolute location of the values.

Given any permutation

$$\sigma : \{1, \ldots, S\} \rightarrow \{1, \ldots, S\},$$

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 require 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.
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.
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.
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.
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 positional encoding

MSE vs Nb. of epochs
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))
>>> 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, 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]])
```

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)
```
With attention, no positional encoding

With attention, positional encoding

MSE

Nb. of epochs

FrançOis Fleuret Deep learning / 13.2. Attention Mechanisms 27 / 29
The end
References
