View code on Github

```python
import math
from typing import Optional, List

import torch
from torch import nn

from labml import tracker
```

Multi-Headed Attention (MHA)

Open in Colab

This is a tutorial/implementation of multi-headed attention from paper Attention Is All You Need in PyTorch. The implementation is inspired from Annotated Transformer.

Here is the training code that uses a basic transformer with MHA for NLP auto-regression.

Here is an experiment implementation that trains a simple transformer.

Prepare for multi-head attention

This module does a linear transformation and splits the vector into given number of heads for multi-head attention. This is used to transform **key**, **query**, and **value** vectors.

```python
class PrepareForMultiHeadAttention(nn.Module):
```

```python
    def __init__(self, d_model: int, heads: int, d_k: int, bias: bool):
        super().__init__()
```

Linear layer for linear transform

```python
        self.linear = nn.Linear(d_model, heads * d_k, bias=bias)
```

Number of heads

```python
        self.heads = heads
```

Number of dimensions in vectors in each head

```python
        self.d_k = d_k
```

```python
    def forward(self, x: torch.Tensor):
```

Input has shape `[seq_len, batch_size, d_model]` or `[batch_size, d_model]` . We apply the linear transformation to the last dimension and split that into the heads.

```python
        head_shape = x.shape[:-1]
```

Linear transform

```python
        x = self.linear(x)
```

Split last dimension into heads

```python
        x = x.view(*head_shape, self.heads, self.d_k)
```

Output has shape `[seq_len, batch_size, heads, d_k]` or `[batch_size, heads, d_model]`

```python
        return x
```

Multi-Head Attention Module

This computes scaled multi-headed attention for given `query` , `key` and `value` vectors.

```python
class MultiHeadAttention(nn.Module):
```

$$Attention(Q, K, V) = \underset{seq}{softmax}\Bigg(\frac{QK^\top}{\sqrt{d_k}}\Bigg)V$$

In simple terms, it finds keys that matches the query, and gets the values of those keys.

It uses dot-product of query and key as the indicator of how matching they are. Before taking the $softmax$ the dot-products are scaled by $\frac{1}{\sqrt{d_k}}$. This is done to avoid large dot-product values causing softmax to give very small gradients when d_k is large.

Softmax is calculated along the axis of of the sequence (or time).

- `heads` is the number of heads.
- `d_model` is the number of features in the `query` , `key` and `value` vectors.

```
    def __init__(self, heads: int, d_model: int, dropout_prob: float = 0.1, bias: bool = True):
```

```
        super().__init__()
```

Number of features per head

```
        self.d_k = d_model // heads
```

Number of heads

```
        self.heads = heads
```

These transform the `query` , `key` and `value` vectors for multi-headed attention.

```
        self.query = PrepareForMultiHeadAttention(d_model, heads, self.d_k, bias=bias)
        self.key = PrepareForMultiHeadAttention(d_model, heads, self.d_k, bias=bias)
        self.value = PrepareForMultiHeadAttention(d_model, heads, self.d_k, bias=True)
```

Softmax for attention along the time dimension of `key`

```
        self.softmax = nn.Softmax(dim=1)
```

Output layer

```
        self.output = nn.Linear(d_model, d_model)
```

Dropout

```
        self.dropout = nn.Dropout(dropout_prob)
```

Scaling factor before the softmax

```
        self.scale = 1 / math.sqrt(self.d_k)
```

We store attentions so that it can be used for logging, or other computations if needed

```
        self.attn = None
```

Calculate scores between queries and keys

This method can be overridden for other variations like relative attention.

```
    def get_scores(self, query: torch.Tensor, key: torch.Tensor):
```

Calculate $QK^\top$ or $S_{ijbh} = \sum_d Q_{ibhd} K_{jbhd}$

```
        return torch.einsum('ibhd,jbhd->ijbh', query, key)
```

`mask` has shape `[seq_len_q, seq_len_k, batch_size]` , where first dimension is the query dimension. If the query dimension is equal to 1 it will be broadcasted.

```
    def prepare_mask(self, mask: torch.Tensor, query_shape: List[int], key_shape: List[int]):
```

```
        assert mask.shape[0] == 1 or mask.shape[0] == query_shape[0]
        assert mask.shape[1] == key_shape[0]
        assert mask.shape[2] == 1 or mask.shape[2] == query_shape[1]
```

Same mask applied to all heads.

```
        mask = mask.unsqueeze(-1)
```

resulting mask has shape `[seq_len_q, seq_len_k, batch_size, heads]`

```
        return mask
```

`query` , `key` and `value` are the tensors that store collection of *query*, *key* and *value* vectors. They have shape `[seq_len, batch_size, d_model]` .

`mask` has shape `[seq_len, seq_len, batch_size]` and `mask[i, j, b]` indicates whether for batch `b` , query at position `i` has access to key-value at position `j` .

```
    def forward(self, *,
                query: torch.Tensor,
                key: torch.Tensor,
                value: torch.Tensor,
                mask: Optional[torch.Tensor] = None):
```

`query` , `key` and `value` have shape `[seq_len, batch_size, d_model]`

```
        seq_len, batch_size, _ = query.shape

        if mask is not None:
            mask = self.prepare_mask(mask, query.shape, key.shape)
```

Prepare `query` , `key` and `value` for attention computation. These will then have shape `[seq_len, batch_size, heads, d_k]` .

```
        query = self.query(query)
        key = self.key(key)
        value = self.value(value)
```

Compute attention scores $QK^\top$. This gives a tensor of shape `[seq_len, seq_len, batch_size, heads]` .

```
        scores = self.get_scores(query, key)
```

Scale scores $\frac{QK^\top}{\sqrt{d_k}}$

```
        scores *= self.scale
```

Apply mask

```
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
```

$softmax$ attention along the key sequence dimension $\underset{seq}{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$

```python
attn = self.softmax(scores)
```

Save attentions if debugging

```python
tracker.debug('attn', attn)
```

Apply dropout

```python
attn = self.dropout(attn)
```

Multiply by values

$$\underset{seq}{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

```python
x = torch.einsum("ijbh,jbhd->ibhd", attn, value)
```

Save attentions for any other calculations

```python
self.attn = attn.detach()
```

Concatenate multiple heads

```python
x = x.reshape(seq_len, batch_size, -1)
```

Output layer

```python
return self.output(x)
```