#A

Transposition Operations

Generate Random PyTorch Tensor:

```
Q = torch.rand((5,4))
Q.shape
torch.Size([5, 4])
```

Perform Transposition Operations:

```
Qa = Q.transpose(0,1)
Qa.shape
torch.Size([4 5])

Qb = Q.transpose(-2,-1)
Qb.shape
torch.Size([4 5])
```

```
Qa - Qb = tensor([[0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0]])
```

Parameters:

```
batch_size = 4
seq_len = 8
d_model = 12
num_heads = 2
d_k = 6
max_seq_length ≡ seq_len = 8
d_ff = 10
num_layers = 3
source_vocab_size = 20
target_vocab_size =
```

Transposition and Viewing Operations #B

```
batch_size = 4
seq_len = 8
d_model = 12
num_heads = 2
d_k = 6

x = torch.rand((batch_size, seq_len, d_model))
x.shape
torch.Size([4, 8, 12])

xV = x.view(batch_size, seq_len, num_heads, d_k)
xV.shape
torch.Size([4, 8, 2, 6])

xT = xV.transpose(1, 2)
xT.shape([4, 2, 8, 6])
```

#5

Instantiate MultiHeadAttention module.

Mind that:

batch_size = 4
seq_len = 8
d_model = 12
num_heads = 2
d_k = 6

```
from classes.MultiHeadAttention import MultiHeadAttention
mha = MultiHeadAttention(d_model, num_heads)
% Set new Q tensor.
Q = torch.rand((batch_size, seq_len, d_model))
K = torch.rand((batch_size, seq_len, d_model))
V = torch.rand((batch_size, seq_len, d_model))
% Compute the attention output.
O = mha.scaled_dot_product_attention(Q, K, V)
```

Q.shape → torch.Size([4, 8, 12])
K.shape → torch.Size([4, 8, 12])
V.shape → torch.Size([4, 8, 12])
O.shape → torch.Size([4, 8, 12])

```
% We could also call:
O = mha.forward(Q, K, V)
```

```
% Make required imports
import torch.nn as nn
% Set the 4 Linear Layers
Qw = nn.Linear(d_model, d_model)
Kw = nn.Linear(d_model, d_model)
Vw = nn.Linear(d_model, d_model)
Ow = nn.Linear(d_model, d_model)
```

Linear(in_features=12, out_features=12, bias=True)

```
% Pass tensors Q, K and V through the corresponding linear layers.
Q = Qw(Q)
K = Kw(K)
V = Vw(V)
```

→ torch.Size([4, 8, 12])
→ torch.Size([4, 8, 12])
→ torch.Size([4, 8, 12])

Checks the MultiHeadAttention

#4

```
% Perform the head splitting operation

Qs = mha.split_heads(Q)     Qs.shape -> torch.Size([4, 2, 8, 6])
Ks = mha.split_heads(K)   } Ks.shape -> torch.Size([4, 2, 8, 6])
Vs = mha.split_heads(V)     Vs.shape -> torch.Size([4, 2, 8, 6])

% Compute the scaled dot product output.

Os = mha.scaled_dot_product_attention(Qs, Ks, Vs) } Os.shape -> torch.Size([4, 2, 8, 6])

% Compute the combined version of O.

Oc = mha.combine_heads(Os) } Oc.shape -> torch.Size([4, 8, 12])

% Compute the final output of the attention layer.

Of = Ow(Oc) } Of.shape -> torch.Size([4, 8, 12])
```

#E

★ Checks PositionEncoding

Mind that:

max-seq-length = 8 = seq-len

d-model = 12

d_ff = 10

```
from classes.PositionEncoding import PositionEncoding
#Set the maximum sequence length equal to the
# existing seq-len parameter.
max_seq_length = seq_len
#Instantiate the PositionalEncoding class
PE = PositionalEncoding(d_model, max_seq_length)
#Sample a random tensor in order to demonstrate
# the forward pass function call.
X = torch.rand((max_seq_length, d_model))  } → X.shape = torch.Size([8, 12])
#Get the new version of X.
X = PE.forward(X)  } → X.shape = torch.Size([1, 8, 12])
```

Checks PositionWiseFeedForward

```
from classes.PositionWiseFeedForward import PositionWiseFeedForward
#Set the internal dimensionality parameter of the position-wise
# feed forward neural network.
d_ff = 10
#Instantiate the PositionWiseFeedForward class.
PWFF = PositionWiseFeedForward(d_model, d_ff)
#Call the corresponding forward pass function.
X = PWFF.forward(X)  } → X.shape = torch.Size([8, 12])
```