Advanced Deep Learning in Computer Vision, Exercises Week 1 submission

Task 1

Code:

```
class Attention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()

        assert embed_dim % num_heads == 0, f'Embedding dimension ({embed_dim}) should be divisible by 
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = self.head_dim ** -0.5

        ####################### insert code here #######################
        self.k_projection = nn.Linear(embed_dim, embed_dim)
        self.q_projection = nn.Linear(embed_dim, embed_dim)
        self.v_projeciton = nn.Linear(embed_dim, embed_dim)
        #################################################################

        self.o_projection = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):

        batch_size, seq_length, embed_dim = x.size()
        keys    = self.k_projection(x)
        queries = self.q_projection(x)
        values  = self.v_projeciton(x)

        # Rearrange keys, queries and values
        # from batch_size x seq_length x embed_dim to (batch_size x num_head) x seq_length x head_dim
        keys = rearrange(keys, 'b s (h d) -> (b h) s d', h=self.num_heads, d=self.head_dim)
        queries = rearrange(queries, 'b s (h d) -> (b h) s d', h=self.num_heads, d=self.head_dim)
        values = rearrange(values, 'b s (h d) -> (b h) s d', h=self.num_heads, d=self.head_dim)

        ####################### insert code here #######################
        ## Compute scaled dot-product attention: (batch_size x num_head) x seq_length x seq_length
        attention = torch.matmul(queries, keys.transpose(1, 2)) * self.scale
        attention = F.softmax(attention, dim=-1)

        ## Apply attention to values: (batch_size x num_head) x seq_length x head_dim
        out = torch.matmul(attention, values)
        #################################################################

        # Rearragne output
        # from (batch_size x num_head) x seq_length x head_dim to batch_size x seq_length x embed_dim
        out = rearrange(out, '(b h) s d -> b s (h d)', h=self.num_heads, d=self.head_dim)

        assert attention.size() == (batch_size*self.num_heads, seq_length, seq_length)
        assert out.size() == (batch_size, seq_length, embed_dim)

        return self.o_projection(out)
```

Output from running test_implementation.py:

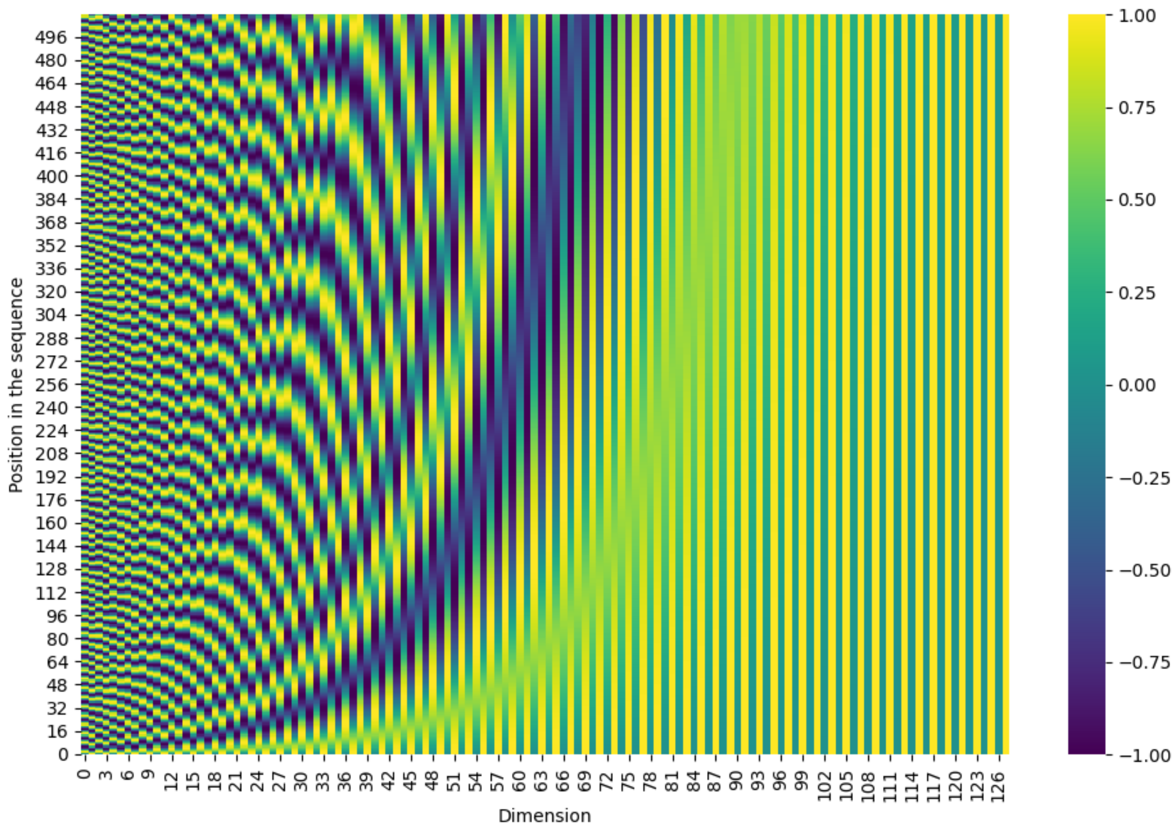
```
test Attention implementation
----------------------------------------------
token shape: torch.Size([16, 512, 128])
output shape: torch.Size([16, 512, 128])
----------------------------------------------
```

Task 2

Code:

```
class PositionalEncoding(nn.Module):
    def __init__(self, embed_dim, max_seq_len=512):

        super(PositionalEncoding, self).__init__()

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_seq_len, embed_dim)
        position = torch.arange(0., max_seq_len).unsqueeze(1)

        ###################### insert code here ######################
        div_term = torch.exp(torch.arange(0., embed_dim, 2) *
                             -(math.log(10000.0) / embed_dim))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        ##############################################################

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        batch_size, seq_length, embed_dim = x.size()
        return x + self.pe[:, :seq_length]
        #return self.dropout(x)
```

Output from running test_implementation.py:



Task 3 & Task 4

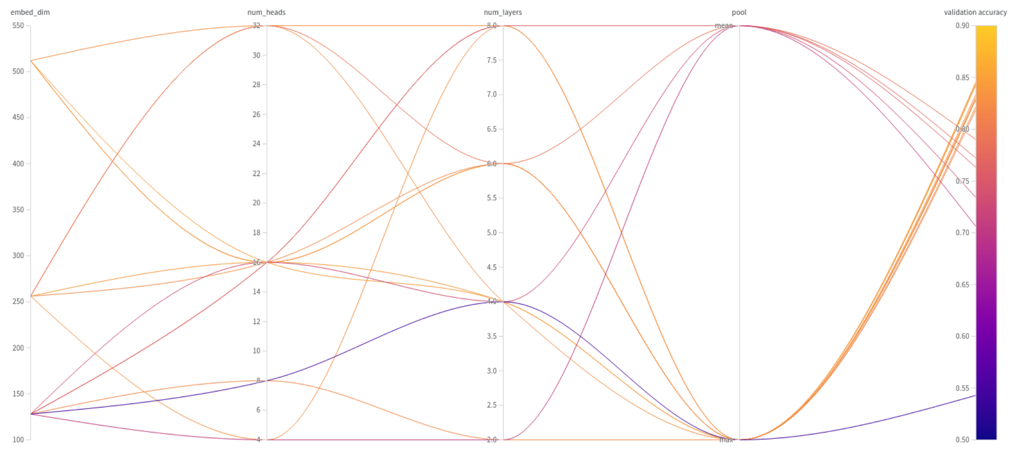
We can affect the model performance on the given task, by changing model parameters. By using Weights and Biases, I have done a random search over the following set of hyperparameter settings:

```
"embed_dim": {'values':[128, 256, 512]},
"num_heads": {'values':[4, 8, 16, 32]},
"num_layers": {'values':[2, 4, 6, 8]},
"pool": {'value':["max", "mean"]},
```

The rest of the hyperparameters were as follows:

```
"num_epochs": {'value':10},
"pos_enc": {'value':"fixed"},#, "learnable"]},
"dropout": {'value':0.0},
"fc_dim": {'value':None},
"batch_size": {'value':16},
"lr": {'value':1e-4},
"warmup_steps": {'value':625},
"weight_decay": {'value':1e-4},
"gradient_clipping": {'value':1},
```

The first search found mainly that better performance was achieved using the max pool option.



I then redid the search with pool being max, and found that having a lower number of heads and higher embed dim made for better performance.