UNetWithAttention Documentation

Niccolò Ferrari

31/07/2024

1 UNetWithAttention Class

1.1 Overview

The `UNetWithAttention` class implements an advanced U-Net architecture augmented with attention mechanisms and configurable skip connections. This class is designed to handle complex image segmentation tasks where attention mechanisms can enhance the model's ability to focus on crucial features, thereby improving the quality of predictions.

1.2 Attributes

- **input_shape** (tuple): The shape of the input image tensor, e.g., $(224, 224, 3)$ for RGB images.
- **timestamp_dim** (int): The dimensionality of the timestamp input, typically a scalar (e.g., 1).
- **filter_list** (list): List of integers representing the number of filters for each encoder and decoder block.
- **num_skip_connections** (int): Number of skip connections from the deepest layer of the encoder.
- **num_heads** (int): Number of attention heads in the multi-head attention layers.
- **key_dim** (int): Dimensionality of the key vectors for the multi-head attention layers.
- **use_bias** (bool): Whether biases are included in convolutional layers and attention mechanisms.
- **activation** (str): Activation function used in the convolutional layers (e.g., 'swish').
- **model** (Model): Keras Model instance representing the complete U-Net with attention architecture.

1.3 Methods

```
                def __init__(self, input_shape, timestamp_dim, filter_list
, num_skip_connections, num_heads=4, key_dim=64, use_bias=False,
activation='swish'):
```

Initializes the `UNetWithAttention` class with specified parameters.

```
                def _conv_block(self, x, filters):
```

Creates a convolutional block with the following operations:

- Convolution: $\text{Conv2D}(x, \text{filters}, (3, 3), \text{padding} =' same')$
- Batch Normalization: $\text{BatchNormalization}(x)$
- Activation: $\text{Activation}(\text{activation})(x)$

The convolution operation can be expressed mathematically as:

$$y = \text{Conv2D}(x) = \text{ReLU}((x * w) + b)$$

where $*$ denotes convolution, w is the filter, and b is the bias [2].

```
def _residual_block(self, x, filters):
```

Constructs a residual block with:

- A residual connection: $\text{Conv2D}(x, \text{filters}, (1, 1))$
- Two convolutional blocks: `_conv_block`(x, filters)
- An addition operation: $x = x + \text{res}$

The residual block can be mathematically represented as:

$$\text{Output} = \text{Activation}(\text{Conv2D}_2(\text{Conv2D}_1(x) + \text{res}))$$

This architecture follows the principles outlined in [1].

```
def _neighborhood_attention(self, query, key, value,
    num_heads, key_dim, neighborhood_size, dropout_rate=0.1):
```

Applies Neighborhood Attention using the MultiHeadAttention mechanism. Key steps include:

- Reshape inputs: $\text{query}, \text{key}, \text{value}$ to shape $[\text{batch\_size}, \text{seq\_len}, \text{depth}]$
- Compute attention mask based on neighborhood size N:

$$\text{Mask}_{ij} = \begin{cases} 1 & \text{if } |i - j| \leq N \\ 0 & \text{otherwise} \end{cases}$$

- Apply MultiHeadAttention: $\text{mha}(\text{query}, \text{key}, \text{value}, \text{attention\_mask})$

The Neighborhood Attention mechanism builds on the principles outlined by [4] and [3].

```
def _multihead_attention_block(self, x):
```

Uses multi-head attention and neighborhood attention:

- Multi-Head Attention: $\text{MultiHeadAttention}(x, x)$
- Neighborhood Attention: `_neighborhood_attention`
- Apply residual and normalization:

$$\text{Output} = \text{LayerNormalization}(\text{Activation}(x + \text{Attn\_Output} + \text{Nttn\_Output}))$$

The use of multi-head attention is based on [4].

```
def _positional_embedding(self, x):
```

Adds positional embeddings to the input tensor x:

- Compute positional embeddings: $\text{pos\_emb} = \text{Embedding}(\text{positions})$
- Add to input tensor: $x + \text{pos\_emb}$

Positional embeddings are derived from [4].

```
def _encoder_block(self, x, filters):
```

Constructs an encoder block:

- Apply residual block: `_residual_block`(x, filters)
- Downsample with convolution: $\text{Conv2D}(x, \text{filters}, (3,3), \text{strides} = (2,2))$

The downsampling operation can be expressed as:

$$x_{\text{down}} = \text{Conv2D}(x, \text{filters}, (3,3), \text{strides} = (2,2))$$

```
def _decoder_block(self, x, skip_features, filters):
```

Constructs a decoder block with:

- Upsample: $\text{Conv2DTranspose}(x, \text{filters}, (3,3), \text{strides} = (2,2))$
- Concatenate with skip connections: $x = \text{Concatenate}([x, \text{skip\_features}])$
- Apply positional embedding and multi-head attention

The upsampling operation is:

$$x_{\text{up}} = \text{Conv2DTranspose}(x, \text{filters}, (3,3), \text{strides} = (2,2))$$

```
def build_model(self):
```

Builds the complete U-Net model with attention mechanisms, including:

- Encoder blocks
- Bottleneck with residual and attention
- Decoder blocks with skip connections
- Output layer with sigmoid activation

```
def print_model(self):
```

Prints the summary of the built model.

```
def save_model_plot(self, filename='model_plot.png'):
```

Saves a visual representation of the model architecture.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[3] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Ilya Polosukhin. Image transformer. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 4055–4064, 2018.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Ilya Polosukhin. Attention is all you need. *Advances in neural information processing systems*, pages 5998–6008, 2017.