"""

Different type of convolution layers to be used in the ClemNet.

"""

from typing import Any

import tensorflow as tf

class KerasConv1D(tf.keras.layers.Layer):

"""

Basic Conv1D layer in a wrapper to be compatible with ClemNet.

"""

def \_\_init\_\_(

self,

kernel\_size: int,

filters: int,

strides: int,

padding: str,

use\_bias: bool = True,

kernel\_initializer: str = "glorot\_uniform",

bias\_initializer: str = "zeros",

\*\*kwargs: Any,

):

super(KerasConv1D, self).\_\_init\_\_(\*\*kwargs)

self.kernel\_size = kernel\_size

self.filters = filters

self.use\_bias = use\_bias

self.kernel\_initializer = kernel\_initializer

self.bias\_initializer = bias\_initializer

self.strides = strides

self.padding = padding

def build(self, input\_shape: tf.TensorShape) -> None:

assert (

len(input\_shape) == 3

), f"Tensor shape must be of length 3. Passed tensor of shape {input\_shape}."

self.features = input\_shape[1]

self.w = tf.keras.layers.Conv1D(

kernel\_size=self.kernel\_size,

filters=self.filters,

strides=self.strides,

padding=self.padding,

use\_bias=self.use\_bias,

kernel\_initializer=self.kernel\_initializer,

bias\_initializer=self.bias\_initializer,

name=self.name,

)

def call(self, inputs: tf.Tensor, \*\*kwargs: Any) -> tf.Tensor:

return self.w(inputs)

class ChannelWiseDense(tf.keras.layers.Layer):

"""

Dense layer is applied to each channel separately. This is more memory and computationally

efficient than flattening the channels and performing single dense layers over it which is the

default behavior in tf1.

"""

def \_\_init\_\_(

self,

output\_size: int,

use\_bias: bool,

kernel\_initializer: str = "uniform\_glorot",

bias\_initializer: str = "zeros",

\*\*kwargs: Any,

):

super(ChannelWiseDense, self).\_\_init\_\_(\*\*kwargs)

self.output\_size = output\_size

self.use\_bias = use\_bias

self.kernel\_initializer = kernel\_initializer

self.bias\_initializer = bias\_initializer

def build(self, input\_shape: tf.TensorShape) -> None:

assert (

len(input\_shape) == 3

), f"Tensor shape must be of length 3. Passed tensor of shape {input\_shape}."

input\_size = input\_shape[1]

channels = input\_shape[2]

self.kernel = self.add\_weight(

name="kernel",

shape=(channels, input\_size, self.output\_size),

initializer=self.kernel\_initializer,

trainable=True,

)

self.bias = self.add\_weight(

name="bias",

shape=(channels, self.output\_size),

initializer=self.bias\_initializer,

trainable=self.use\_bias,

)

def call(self, inputs: tf.Tensor, \*\*kwargs: Any) -> tf.Tensor:

x = inputs

transposed\_x = tf.transpose(x, perm=[2, 0, 1])

transposed\_residual = (

tf.transpose(tf.matmul(transposed\_x, self.kernel), perm=[1, 0, 2]) + self.bias

)

output = tf.transpose(transposed\_residual, perm=[0, 2, 1])

return output

class ResidualLayer(tf.keras.layers.Layer):

"""

Layer implementing a 3D-residual connection.

"""

def build(self, input\_shape: tf.TensorShape) -> None:

assert (

len(input\_shape) == 3

), f"Tensor shape must be of length 3. Passed tensor of shape {input\_shape}."

def call(self, inputs: tf.Tensor, residual: tf.Tensor, \*\*kwargs: Any) -> tf.Tensor:

shortcut = tf.keras.layers.Conv1D(

filters=int(residual.shape[2]), strides=1, kernel\_size=1, padding="SAME", use\_bias=False

)(inputs)

output = tf.add(shortcut, residual)

return output