"""Code to construct the VLAAI network.

Code was extrcted from https://github.com/exporl/vlaai

"""

import tensorflow as tf

def extractor(

filters=(256, 256, 256, 128, 128),

kernels=(64,) \* 5,

dilation\_rate = 1,

input\_channels=64,

normalization\_fn=lambda x: tf.keras.layers.LayerNormalization()(x),

activation\_fn=lambda x: tf.keras.layers.LeakyReLU()(x),

name="extractor",

):

"""Construct the extractor model.

Parameters

----------

filters: Sequence[int]python

Number of filters for each layer.

kernels: Sequence[int]

Kernel size for each layer.

input\_channels: int

Number of EEG channels in the input

normalization\_fn: Callable[[tf.Tensor], tf.Tensor]

Function to normalize the contents of a tensor.

activation\_fn: Callable[[tf.Tensor], tf.Tensor]

Function to apply an activation function to the contents of a tensor.

name: str

Name of the model.

Returns

-------

tf.keras.models.Model

The extractor model.

"""

eeg = tf.keras.layers.Input((None, input\_channels))

x = eeg

if len(filters) != len(kernels):

raise ValueError("'filters' and 'kernels' must have the same length")

# Add the convolutional layers

i = 0

for filter\_, kernel in zip(filters, kernels):

i +=1

if i == len(filters) :

padding = 'valid'

else:

padding = 'valid'

x = tf.keras.layers.Conv1D(filter\_, kernel, dilation\_rate=dilation\_rate,padding=padding )(x)

x = normalization\_fn(x)

x = activation\_fn(x)

x = tf.keras.layers.ZeroPadding1D((0, kernel - 1))(x)

return tf.keras.models.Model(inputs=[eeg], outputs=[x], name=name)

def output\_context(

filter\_=64,

kernel=64,

input\_channels=64,

normalization\_fn=lambda x: tf.keras.layers.LayerNormalization()(x),

activation\_fn=lambda x: tf.keras.layers.LeakyReLU()(x),

name="output\_context\_model",

):

"""Construct the output context model.

Parameters

----------

filter\_: int

Number of filters for the convolutional layer.

kernel: int

Kernel size for the convolutional layer.

input\_channels: int

Number of EEG channels in the input.

normalization\_fn: Callable[[tf.Tensor], tf.Tensor]

Function to normalize the contents of a tensor.

activation\_fn: Callable[[tf.Tensor], tf.Tensor]

Function to apply an activation function to the contents of a tensor.

name: str

Name of the model.

Returns

-------

tf.keras.models.Model

The output context model.

"""

inp = tf.keras.layers.Input((None, input\_channels))

x = tf.keras.layers.ZeroPadding1D((kernel - 1, 0))(inp)

x = tf.keras.layers.Conv1D(filter\_, kernel)(x)

x = normalization\_fn(x)

x = activation\_fn(x)

return tf.keras.models.Model(inputs=[inp], outputs=[x], name=name)

def vlaai(

nb\_blocks=4,

extractor\_model=None,

output\_context\_model=None,

use\_skip=True,

input\_channels=64,

output\_dim=1,

name="vlaai",

):

"""Construct the VLAAI model.

Parameters

----------

nb\_blocks: int

Number of repeated blocks to use.

extractor\_model: Callable[[tf.Tensor], tf.Tensor]

The extractor model to use.

output\_context\_model: Callable[[tf.Tensor], tf.Tensor]

The output context model to use.

use\_skip: bool

Whether to use skip connections.

input\_channels: int

Number of EEG channels in the input.

output\_dim: int

Number of output dimensions.

name: str

Name of the model.

Returns

-------

tf.keras.models.Model

The VLAAI model.

"""

if extractor\_model is None:

extractor\_model = extractor()

if output\_context\_model is None:

output\_context\_model = output\_context()

eeg = tf.keras.layers.Input((None, input\_channels))

# If using skip connections: start with x set to zero

if use\_skip:

x = tf.zeros\_like(eeg)

else:

x = eeg

# Iterate over the blocks

for i in range(nb\_blocks):

if use\_skip:

x = extractor\_model(eeg + x)

else:

x = extractor\_model(x)

x = tf.keras.layers.Dense(input\_channels)(x)

x = output\_context\_model(x)

x = tf.keras.layers.Dense(output\_dim)(x)

return tf.keras.models.Model(inputs=[eeg], outputs=[x], name=name)

def pearson\_tf(y\_true, y\_pred, axis=1):

"""Pearson correlation function implemented in tensorflow.

Parameters

----------

y\_true: tf.Tensor

Ground truth labels. Shape is (batch\_size, time\_steps, n\_features)

y\_pred: tf.Tensor

Predicted labels. Shape is (batch\_size, time\_steps, n\_features)

axis: int

Axis along which to compute the pearson correlation. Default is 1.

Returns

-------

tf.Tensor

Pearson correlation.

Shape is (batch\_size, 1, n\_features) if axis is 1.

"""

# Compute the mean of the true and predicted values

y\_true\_mean = tf.reduce\_mean(y\_true, axis=axis, keepdims=True)

y\_pred\_mean = tf.reduce\_mean(y\_pred, axis=axis, keepdims=True)

# Compute the numerator and denominator of the pearson correlation

numerator = tf.reduce\_sum(

(y\_true - y\_true\_mean) \* (y\_pred - y\_pred\_mean),

axis=axis,

keepdims=True,

)

std\_true = tf.reduce\_sum(tf.square(y\_true - y\_true\_mean), axis=axis, keepdims=True)

std\_pred = tf.reduce\_sum(tf.square(y\_pred - y\_pred\_mean), axis=axis, keepdims=True)

denominator = tf.sqrt(std\_true \* std\_pred)

# Compute the pearson correlation

return tf.reduce\_mean(tf.math.divide\_no\_nan(numerator, denominator), axis=-1)

def pearson\_tf\_non\_averaged(y\_true, y\_pred, axis=1):

"""Pearson correlation function implemented in tensorflow.

Parameters

----------

y\_true: tf.Tensor

Ground truth labels. Shape is (batch\_size, time\_steps, n\_features)

y\_pred: tf.Tensor

Predicted labels. Shape is (batch\_size, time\_steps, n\_features)

axis: int

Axis along which to compute the pearson correlation. Default is 1.

Returns

-------

tf.Tensor

Pearson correlation.

Shape is (batch\_size, 1, n\_features) if axis is 1.

"""

# Compute the mean of the true and predicted values

y\_true\_mean = tf.reduce\_mean(y\_true, axis=axis, keepdims=True)

y\_pred\_mean = tf.reduce\_mean(y\_pred, axis=axis, keepdims=True)

# Compute the numerator and denominator of the pearson correlation

numerator = tf.reduce\_sum(

(y\_true - y\_true\_mean) \* (y\_pred - y\_pred\_mean),

axis=axis,

keepdims=True,

)

std\_true = tf.reduce\_sum(tf.square(y\_true - y\_true\_mean), axis=axis, keepdims=True)

std\_pred = tf.reduce\_sum(tf.square(y\_pred - y\_pred\_mean), axis=axis, keepdims=True)

denominator = tf.sqrt(std\_true \* std\_pred)

# Compute the pearson correlation

return tf.math.divide\_no\_nan(numerator, denominator)

@tf.function

def pearson\_loss(y\_true, y\_pred, axis=1):

"""Pearson loss function.

Parameters

----------

y\_true: tf.Tensor

True values. Shape is (batch\_size, time\_steps, n\_features)

y\_pred: tf.Tensor

Predicted values. Shape is (batch\_size, time\_steps, n\_features)

Returns

-------

tf.Tensor

Pearson loss.

Shape is (batch\_size, 1, n\_features)

"""

return -pearson\_tf(y\_true, y\_pred, axis=axis)

@tf.function

def pearson\_metric(y\_true, y\_pred, axis=1):

"""Pearson metric function.

Parameters

----------

y\_true: tf.Tensor

True values. Shape is (batch\_size, time\_steps, n\_features)

y\_pred: tf.Tensor

Predicted values. Shape is (batch\_size, time\_steps, n\_features)

Returns

-------

tf.Tensor

Pearson metric.

Shape is (batch\_size, 1, n\_features)

"""

return pearson\_tf(y\_true, y\_pred, axis=axis)