"""

Implementing Sequential Layer container

"""

from .layer import Layer

from tensorflow import keras

from tensorflow.python.layers import base

class Sequential(Layer):

"""

A sequential stack of layers.

Arguments:

layers: list of layers to add to the model.

Output:

the output of the sequential layers

"""

def \_\_init\_\_(self, layers=None, \*\*kwargs):

self.\_layers = [] # Stack of layers.

self.\_layer\_names = [] # Stack of layers names

self.\_layer\_outputs = []

# Add to the model any layers passed to the constructor.

if layers:

for layer in layers:

self.add(layer)

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

def add(self, layer):

"""Adds a layer instance on top of the layer stack.

Arguments:

layer:

layer instance.

Raises:

TypeError:

if the layer argument is not instance of base.Layer

"""

if not isinstance(layer, base.Layer) and not isinstance(layer, keras.layers.Layer):

raise TypeError('The added layer must be an instance of class Layer')

if layer.name in self.\_layer\_names:

raise ValueError('Layer with name %s already exists in sequential layer' % layer.name)

self.\_layers.append(layer)

self.\_layer\_names.append(layer.name)

def pop(self):

"""Removes the last layer in the model.

Raises:

TypeError:

if there are no layers in the model.

"""

if not self.\_layers or not self.\_layer\_names:

raise TypeError('There are no layers in the model.')

self.\_layers.pop()

self.\_layer\_names.pop()

def call(self, inputs, \*\*kwargs): # pylint: disable=unused-argument

"""The logic of the layer lives here.

Arguments:

inputs:

input tensor(s).

Returns:

The output of the sequential layers

"""

self.\_layer\_outputs = []

for layer in self.\_layers:

# don't use layer.call because you want to build individual layers

inputs = layer(inputs) # overwrites the current input after it has been processed

self.\_layer\_outputs.append(inputs)

return inputs

@property

def layers(self):

""" Return the layers in the sequential layer """

return self.\_layers

@property

def layer\_names(self):

""" Return the layer names in the sequential layer """

return self.\_layer\_names

@property

def layer\_outputs(self):

""" Return the layer outputs in the sequential layer """

return self.\_layer\_outputs

def get(self, key):

"""Retrieves the n-th layer.

Arguments:

key:

index of the layer

Output:

The n-th layer where n is equal to the key.

"""

return self.\_layers[key]

def get\_output(self, key):

"""Retrieves the n-th layer output.

Arguments:

key:

index of the layer

Output:

The intermediary output equivalent to the nth layer, where n is equal to the key.

"""

return self.\_layer\_outputs[key]

def get\_layer\_by\_name(self, name):

"""Retrieves the layer corresponding to the name.

Arguments:

name:

name of the layer

Output:

list of layers that have the name desired

"""

return self.\_layers[self.\_layer\_names.index(name)]

def get\_layer\_output\_by\_name(self, name):

"""Retrieves the layer output corresponding to the name.

Arguments:

name:

name of the layer

Output:

list of the output of the layers that have the desired name

"""

return self.\_layer\_outputs[self.\_layer\_names.index(name)]

@property

def init(self):

""" returns a list of initialization ops (one per layer) """

return [layer.init for layer in self.\_layers]

def compute\_output\_shape(self, input\_shape):

"""Computes the output shape of the layer given the input shape.

Args:

input\_shape: A (possibly nested tuple of) `TensorShape`. It need not

be fully defined (e.g. the batch size may be unknown).

Raise NotImplementedError.

"""

raise NotImplementedError