from \_\_future\_\_ import absolute\_import

import warnings

from tensorflow.keras import backend as K

from tensorflow.keras import activations

from tensorflow.keras import initializers

from tensorflow.keras import regularizers

from tensorflow.keras import constraints

from tensorflow.keras.layers import Layer

from tensorflow.keras.layers import InputSpec

#from tensorflow.keras.legacy import interfaces

from tensorflow.keras.layers import RNN

#from tensorflow.keras.layers.recurrent import \_generate\_dropout\_mask

class IndRNNCell(Layer):

*""" NOTE: This package is sourced from the link below, but all dependencies have been reworked to use*

*tensorflow.keras rather than Keras (due to not having permission to install Keras on Cuda)*

*"""*

"""Independently Recurrent Neural Networks Cell class.

Derived from the paper [Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN](https://arxiv.org/abs/1803.04831)

Ref: [Tensorflow implementation](https://github.com/batzner/indrnn)

# Arguments

units: Positive integer, dimensionality of the output space.

recurrent\_clip\_min: Can be one of None, -1 or float.

If None, clipping of weights will not take place.

If float, exact value will be used as clipping range

If -1, will calculate the clip value for `relu` activation

recurrent\_clip\_max: Can be one of None or float.

If None, clipping of weights will not take place.

If float, exact value will be used as clipping range

If -1, will calculate the clip value for `relu` activation

activation: Activation function to use

(see [activations](../activations.md)).

If you pass None, no activation is applied

(ie. "linear" activation: `a(x) = x`).

use\_bias: Boolean, whether the layer uses a bias vector.

kernel\_initializer: Initializer for the `kernel` weights matrix,

used for the linear transformation of the inputs

(see [initializers](../initializers.md)).

recurrent\_initializer: Initializer for the `recurrent\_kernel`

weights matrix, used for the linear transformation of the

recurrent state.

Can be `None` or an available initializer. Defaults to `None`.

If None, defaults to uniform initialization.

If None, and recurrent\_clip\_min/max is not None, then

it uses those clip values as for uniform initialization.

(see [initializers](../initializers.md)).

bias\_initializer: Initializer for the bias vector

(see [initializers](../initializers.md)).

kernel\_regularizer: Regularizer function applied to

the `kernel` weights matrix

(see [regularizer](../regularizers.md)).

recurrent\_regularizer: Regularizer function applied to

the `recurrent\_kernel` weights matrix

(see [regularizer](../regularizers.md)).

bias\_regularizer: Regularizer function applied to the bias vector

(see [regularizer](../regularizers.md)).

kernel\_constraint: Constraint function applied to

the `kernel` weights matrix

(see [constraints](../constraints.md)).

recurrent\_constraint: Constraint function applied to

the `recurrent\_kernel` weights matrix

(see [constraints](../constraints.md)).

bias\_constraint: Constraint function applied to the bias vector

(see [constraints](../constraints.md)).

dropout: Float between 0 and 1.

Fraction of the units to drop for

the linear transformation of the inputs.

recurrent\_dropout: Float between 0 and 1.

Fraction of the units to drop for

the linear transformation of the recurrent state.

implementation: Implementation mode, must be 2.

Mode 1 will structure its operations as a larger number of

smaller dot products and additions, whereas mode 2 will

batch them into fewer, larger operations. These modes will

have different performance profiles on different hardware and

for different applications.

"""

def \_\_init\_\_(self, units,

recurrent\_clip\_min=-1,

recurrent\_clip\_max=-1,

activation='relu',

use\_bias=True,

kernel\_initializer='glorot\_uniform',

recurrent\_initializer=None,

bias\_initializer='zeros',

kernel\_regularizer=None,

recurrent\_regularizer=None,

bias\_regularizer=None,

kernel\_constraint=None,

recurrent\_constraint=None,

bias\_constraint=None,

dropout=0.,

recurrent\_dropout=0.,

implementation=2,

\*\*kwargs):

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

if implementation != 2:

warnings.warn(

"IndRNN only supports implementation 2 for the moment. Defaulting to implementation = 2")

implementation = 2

if recurrent\_clip\_min is None or recurrent\_clip\_max is None:

recurrent\_clip\_min = None

recurrent\_clip\_max = None

self.units = units

self.recurrent\_clip\_min = recurrent\_clip\_min

self.recurrent\_clip\_max = recurrent\_clip\_max

self.activation = activations.get(activation)

self.use\_bias = use\_bias

self.kernel\_initializer = initializers.get(kernel\_initializer)

self.recurrent\_initializer = initializers.get(recurrent\_initializer) \

if recurrent\_initializer is not None else None

self.bias\_initializer = initializers.get(bias\_initializer)

self.kernel\_regularizer = regularizers.get(kernel\_regularizer)

self.recurrent\_regularizer = regularizers.get(recurrent\_regularizer)

self.bias\_regularizer = regularizers.get(bias\_regularizer)

self.kernel\_constraint = constraints.get(kernel\_constraint)

self.recurrent\_constraint = constraints.get(recurrent\_constraint)

self.bias\_constraint = constraints.get(bias\_constraint)

self.dropout = min(1., max(0., dropout))

self.recurrent\_dropout = min(1., max(0., recurrent\_dropout))

self.implementation = implementation

self.state\_size = (self.units,)

self.\_dropout\_mask = None

self.\_recurrent\_masks = None

def build(self, input\_shape):

input\_dim = input\_shape[-1]

if self.recurrent\_clip\_min == -1 or self.recurrent\_clip\_max == -1:

self.recurrent\_clip\_min = 0.0

if hasattr(self, 'timesteps') and self.timesteps is not None:

self.recurrent\_clip\_max = pow(2.0, 1. / self.timesteps)

else:

warnings.warn("IndRNNCell: Number of timesteps could not be determined. \n"

"Defaulting to max clipping range of 1.0. \n"

"If this model was trained using a specific timestep during training, "

"inference may be wrong due to this default setting.\n"

"Please ensure that you use the same number of timesteps during training "

"and evaluation")

self.recurrent\_clip\_max = 1.0

self.kernel = self.add\_weight(shape=(input\_dim, self.units),

name='input\_kernel',

initializer=self.kernel\_initializer,

regularizer=self.kernel\_regularizer,

constraint=self.kernel\_constraint)

if self.recurrent\_initializer is None:

if self.recurrent\_clip\_min is not None and self.recurrent\_clip\_max is not None:

initialization\_value = min(self.recurrent\_clip\_max, 1.0)

self.recurrent\_initializer = initializers.uniform(-initialization\_value,

initialization\_value)

else:

self.recurrent\_initializer = initializers.uniform(-1.0, 1.0)

self.recurrent\_kernel = self.add\_weight(shape=(self.units,),

name='recurrent\_kernel',

initializer=self.recurrent\_initializer,

regularizer=self.recurrent\_regularizer,

constraint=self.recurrent\_constraint)

if self.recurrent\_clip\_min is not None and self.recurrent\_clip\_max is not None:

if abs(self.recurrent\_clip\_min):

abs\_recurrent\_kernel = K.abs(self.recurrent\_kernel)

min\_recurrent\_kernel = K.maximum(abs\_recurrent\_kernel, abs(self.recurrent\_clip\_min))

self.recurrent\_kernel = K.sign(self.recurrent\_kernel) \* min\_recurrent\_kernel

self.recurrent\_kernel = K.clip(self.recurrent\_kernel,

self.recurrent\_clip\_min,

self.recurrent\_clip\_max)

if self.use\_bias:

bias\_initializer = self.bias\_initializer

self.bias = self.add\_weight(shape=(self.units,),

name='bias',

initializer=bias\_initializer,

regularizer=self.bias\_regularizer,

constraint=self.bias\_constraint)

else:

self.bias = None

self.built = True

def \_generate\_dropout\_mask(ones, rate, training=None, count=1):

def dropped\_inputs():

return K.dropout(ones, rate)

if count > 1:

return [

K.in\_train\_phase(dropped\_inputs, ones, training=training)

for \_ in range(count)

]

return K.in\_train\_phase(dropped\_inputs, ones, training=training)

def call(self, inputs, states, training=None):

if 0 < self.dropout < 1 and self.\_dropout\_mask is None:

self.\_dropout\_mask = \_generate\_dropout\_mask(

K.ones\_like(inputs),

self.dropout,

training=training,

count=1)

if (0 < self.recurrent\_dropout < 1 and

self.\_recurrent\_masks is None):

\_recurrent\_mask = \_generate\_dropout\_mask(

K.ones\_like(states[0]),

self.recurrent\_dropout,

training=training,

count=1)

self.\_recurrent\_masks = \_recurrent\_mask

# dropout matrices for input units

dp\_mask = self.\_dropout\_mask

# dropout matrices for recurrent units

rec\_dp\_masks = self.\_recurrent\_masks

h\_tm1 = states[0] # previous state

if 0. < self.dropout < 1.:

inputs \*= dp\_mask[0]

if 0. < self.recurrent\_dropout < 1.:

h\_tm1 \*= rec\_dp\_masks[0]

h = K.dot(inputs, self.kernel)

h = h + (h\_tm1 \* self.recurrent\_kernel)

if self.use\_bias:

h = K.bias\_add(h, self.bias)

h = self.activation(h)

if 0 < self.dropout + self.recurrent\_dropout:

if training is None:

h.\_uses\_learning\_phase = True

return h, [h]

def get\_config(self):

config = {'units': self.units,

'recurrent\_clip\_min': self.recurrent\_clip\_min,

'recurrent\_clip\_max': self.recurrent\_clip\_max,

'activation': activations.serialize(self.activation),

'use\_bias': self.use\_bias,

'kernel\_initializer': initializers.serialize(self.kernel\_initializer),

'recurrent\_initializer': initializers.serialize(self.recurrent\_initializer),

'bias\_initializer': initializers.serialize(self.bias\_initializer),

'kernel\_regularizer': regularizers.serialize(self.kernel\_regularizer),

'recurrent\_regularizer': regularizers.serialize(self.recurrent\_regularizer),

'bias\_regularizer': regularizers.serialize(self.bias\_regularizer),

'kernel\_constraint': constraints.serialize(self.kernel\_constraint),

'recurrent\_constraint': constraints.serialize(self.recurrent\_constraint),

'bias\_constraint': constraints.serialize(self.bias\_constraint),

'dropout': self.dropout,

'recurrent\_dropout': self.recurrent\_dropout,

'implementation': self.implementation}

base\_config = super(IndRNNCell, self).get\_config()

return dict(list(base\_config.items()) + list(config.items()))

class IndRNN(RNN):

*"""Independently Recurrent Neural Networks Cell class.*

*Derived from the paper [Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN](https://arxiv.org/abs/1803.04831)*

*Ref: [Tensorflow implementation](https://github.com/batzner/indrnn)*

*# Arguments*

*units: Positive integer, dimensionality of the output space.*

*recurrent\_clip\_min: Can be one of None, -1 or float.*

*If None, clipping of weights will not take place.*

*If float, exact value will be used as clipping range*

*If -1, computes the default clipping range for Relu activations*

*recurrent\_clip\_max: Can be one of None, -1 or float.*

*If None, clipping of weights will not take place.*

*If float, exact value will be used as clipping range*

*If -1, computes the default clipping range for Relu activations*

*activation: Activation function to use*

*(see [activations](../activations.md)).*

*If you pass None, no activation is applied*

*(ie. "linear" activation: `a(x) = x`).*

*use\_bias: Boolean, whether the layer uses a bias vector.*

*kernel\_initializer: Initializer for the `kernel` weights matrix,*

*used for the linear transformation of the inputs.*

*(see [initializers](../initializers.md)).*

*recurrent\_initializer: Initializer for the `recurrent\_kernel`*

*weights matrix,*

*used for the linear transformation of the recurrent state.*

*(see [initializers](../initializers.md)).*

*bias\_initializer: Initializer for the bias vector*

*(see [initializers](../initializers.md)).*

*unit\_forget\_bias: Boolean.*

*If True, add 1 to the bias of the forget gate at initialization.*

*Setting it to true will also force `bias\_initializer="zeros"`.*

*This is recommended in [Jozefowicz et al.](http://www.jmlr.org/proceedings/papers/v37/jozefowicz15.pdf)*

*kernel\_regularizer: Regularizer function applied to*

*the `kernel` weights matrix*

*(see [regularizer](../regularizers.md)).*

*recurrent\_regularizer: Regularizer function applied to*

*the `recurrent\_kernel` weights matrix*

*(see [regularizer](../regularizers.md)).*

*bias\_regularizer: Regularizer function applied to the bias vector*

*(see [regularizer](../regularizers.md)).*

*activity\_regularizer: Regularizer function applied to*

*the output of the layer (its "activation").*

*(see [regularizer](../regularizers.md)).*

*kernel\_constraint: Constraint function applied to*

*the `kernel` weights matrix*

*(see [constraints](../constraints.md)).*

*recurrent\_constraint: Constraint function applied to*

*the `recurrent\_kernel` weights matrix*

*(see [constraints](../constraints.md)).*

*bias\_constraint: Constraint function applied to the bias vector*

*(see [constraints](../constraints.md)).*

*dropout: Float between 0 and 1.*

*Fraction of the units to drop for*

*the linear transformation of the inputs.*

*recurrent\_dropout: Float between 0 and 1.*

*Fraction of the units to drop for*

*the linear transformation of the recurrent state.*

*implementation: Implementation mode, either 1 or 2.*

*Mode 1 will structure its operations as a larger number of*

*smaller dot products and additions, whereas mode 2 will*

*batch them into fewer, larger operations. These modes will*

*have different performance profiles on different hardware and*

*for different applications.*

*return\_sequences: Boolean. Whether to return the last output.*

*in the output sequence, or the full sequence.*

*return\_state: Boolean. Whether to return the last state*

*in addition to the output.*

*go\_backwards: Boolean (default False).*

*If True, process the input sequence backwards and return the*

*reversed sequence.*

*stateful: Boolean (default False). If True, the last state*

*for each sample at index i in a batch will be used as initial*

*state for the sample of index i in the following batch.*

*unroll: Boolean (default False).*

*If True, the network will be unrolled,*

*else a symbolic loop will be used.*

*Unrolling can speed-up a RNN,*

*although it tends to be more memory-intensive.*

*Unrolling is only suitable for short sequences.*

*# References*

*- [Learning to forget: Continual prediction with NestedLSTM](http://www.mitpressjournals.org/doi/pdf/10.1162/089976600300015015)*

*- [Supervised sequence labeling with recurrent neural networks](http://www.cs.toronto.edu/~graves/preprint.pdf)*

*- [A Theoretically Grounded Application of Dropout in Recurrent Neural Networks](http://arxiv.org/abs/1512.05287)*

*- [Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN](https://arxiv.org/abs/1803.04831)*

*"""*

#@interfaces.legacy\_recurrent\_support

def \_\_init\_\_(self, units,

recurrent\_clip\_min=-1,

recurrent\_clip\_max=-1,

activation='relu',

use\_bias=True,

kernel\_initializer='glorot\_uniform',

recurrent\_initializer=None,

bias\_initializer='zeros',

kernel\_regularizer=None,

recurrent\_regularizer=None,

bias\_regularizer=None,

activity\_regularizer=None,

kernel\_constraint=None,

recurrent\_constraint=None,

bias\_constraint=None,

dropout=0.,

recurrent\_dropout=0.,

implementation=2,

return\_sequences=False,

return\_state=False,

go\_backwards=False,

stateful=False,

unroll=False,

\*\*kwargs):

if implementation == 0:

warnings.warn('`implementation=0` has been deprecated, '

'and now defaults to `implementation=2`.'

'Please update your layer call.')

if K.backend() == 'theano':

warnings.warn(

'RNN dropout is no longer supported with the Theano backend '

'due to technical limitations. '

'You can either set `dropout` and `recurrent\_dropout` to 0, '

'or use the TensorFlow backend.')

dropout = 0.

recurrent\_dropout = 0.

cell = IndRNNCell(units,

recurrent\_clip\_min=recurrent\_clip\_min,

recurrent\_clip\_max=recurrent\_clip\_max,

activation=activation,

use\_bias=use\_bias,

kernel\_initializer=kernel\_initializer,

recurrent\_initializer=recurrent\_initializer,

bias\_initializer=bias\_initializer,

kernel\_regularizer=kernel\_regularizer,

recurrent\_regularizer=recurrent\_regularizer,

bias\_regularizer=bias\_regularizer,

kernel\_constraint=kernel\_constraint,

recurrent\_constraint=recurrent\_constraint,

bias\_constraint=bias\_constraint,

dropout=dropout,

recurrent\_dropout=recurrent\_dropout,

implementation=implementation)

super(IndRNN, self).\_\_init\_\_(cell,

return\_sequences=return\_sequences,

return\_state=return\_state,

go\_backwards=go\_backwards,

stateful=stateful,

unroll=unroll,

\*\*kwargs)

self.activity\_regularizer = regularizers.get(activity\_regularizer)

def build(self, input\_shape):

timesteps = input\_shape[1]

if timesteps is None:

warnings.warn("Number of timesteps was not provided. If this model is being used for training purposes, \n"

"it is recommended to provide a finite number of timesteps when defining the input shape, \n"

"so as to initialize the weights of the recurrent kernel properly and avoid exploding gradients.")

self.cell.timesteps = timesteps

super(IndRNN, self).build(input\_shape)

def call(self, inputs, mask=None, training=None, initial\_state=None, constants=None):

self.cell.\_dropout\_mask = None

self.cell.\_recurrent\_masks = None

return super(IndRNN, self).call(inputs,

mask=mask,

training=training,

initial\_state=initial\_state,

constants=constants)

@property

def units(self):

return self.cell.units

@property

def recurrent\_clip\_min(self):

return self.cell.recurrent\_clip\_min

@property

def recurrent\_clip\_max(self):

return self.cell.recurrent\_clip\_max

@property

def activation(self):

return self.cell.activation

@property

def use\_bias(self):

return self.cell.use\_bias

@property

def kernel\_initializer(self):

return self.cell.kernel\_initializer

@property

def recurrent\_initializer(self):

return self.cell.recurrent\_initializer

@property

def bias\_initializer(self):

return self.cell.bias\_initializer

@property

def kernel\_regularizer(self):

return self.cell.kernel\_regularizer

@property

def recurrent\_regularizer(self):

return self.cell.recurrent\_regularizer

@property

def bias\_regularizer(self):

return self.cell.bias\_regularizer

@property

def kernel\_constraint(self):

return self.cell.kernel\_constraint

@property

def recurrent\_constraint(self):

return self.cell.recurrent\_constraint

@property

def bias\_constraint(self):

return self.cell.bias\_constraint

@property

def dropout(self):

return self.cell.dropout

@property

def recurrent\_dropout(self):

return self.cell.recurrent\_dropout

@property

def implementation(self):

return self.cell.implementation

def get\_config(self):

config = {'units': self.units,

'recurrent\_clip\_min': self.recurrent\_clip\_min,

'recurrent\_clip\_max': self.recurrent\_clip\_max,

'activation': activations.serialize(self.activation),

'use\_bias': self.use\_bias,

'kernel\_initializer': initializers.serialize(self.kernel\_initializer),

'recurrent\_initializer': initializers.serialize(self.recurrent\_initializer),

'bias\_initializer': initializers.serialize(self.bias\_initializer),

'kernel\_regularizer': regularizers.serialize(self.kernel\_regularizer),

'recurrent\_regularizer': regularizers.serialize(self.recurrent\_regularizer),

'bias\_regularizer': regularizers.serialize(self.bias\_regularizer),

'activity\_regularizer': regularizers.serialize(self.activity\_regularizer),

'kernel\_constraint': constraints.serialize(self.kernel\_constraint),

'recurrent\_constraint': constraints.serialize(self.recurrent\_constraint),

'bias\_constraint': constraints.serialize(self.bias\_constraint),

'dropout': self.dropout,

'recurrent\_dropout': self.recurrent\_dropout,

'implementation': self.implementation}

base\_config = super(IndRNN, self).get\_config()

del base\_config['cell']

return dict(list(base\_config.items()) + list(config.items()))

@classmethod

def from\_config(cls, config):

if 'implementation' in config and config['implementation'] == 0:

config['implementation'] = 2

return cls(\*\*config)