from twitter.deepbird.compat.v1.rnn import stack\_bidirectional\_dynamic\_rnn

import tensorflow.compat.v1 as tf

import tensorflow

import twml

def \_get\_rnn\_cell\_creator(cell\_type):

if cell\_type == "LSTM":

Cell = tf.nn.rnn\_cell.LSTMCell

elif cell\_type == "GRU":

Cell = tf.nn.rnn\_cell.GRUCell

else:

raise ValueError("cell\_type: %s is not supported."

"It should be one of 'LSTM' or 'GRU'." % cell\_type)

return Cell

def \_apply\_dropout\_wrapper(rnn\_cells, dropout):

""" Apply dropout wrapper around each cell if necessary """

if rnn\_cells is None:

return None

cells = []

for i, dropout\_rate in enumerate(dropout):

cell = rnn\_cells[i]

if dropout\_rate > 0:

cell = tf.nn.rnn\_cell.DropoutWrapper(cell, input\_keep\_prob=(1.0 - dropout\_rate))

cells.append(cell)

return cells

def \_create\_bidirectional\_rnn\_cell(num\_units, dropout, cell\_type):

scope\_name = "lstm" if cell\_type else "gru"

with tf.variable\_scope(scope\_name):

Cell = \_get\_rnn\_cell\_creator(cell\_type)

cells\_forward = [Cell(output\_size) for output\_size in num\_units]

cells\_backward = [Cell(output\_size) for output\_size in num\_units]

cells\_forward = \_apply\_dropout\_wrapper(cells\_forward, dropout)

cells\_backward = \_apply\_dropout\_wrapper(cells\_backward, dropout)

def stacked\_rnn\_cell(inputs, sequence\_lengths):

with tf.variable\_scope(scope\_name):

outputs, final\_states, \_ = stack\_bidirectional\_dynamic\_rnn(

cells\_fw=cells\_forward, cells\_bw=cells\_backward, inputs=inputs,

sequence\_length=sequence\_lengths, dtype=inputs.dtype)

return final\_states[-1][-1]

return stacked\_rnn\_cell

def \_create\_unidirectional\_rnn\_cell(num\_units, dropout, cell\_type):

scope\_name = "lstm" if cell\_type else "gru"

with tf.variable\_scope(scope\_name):

Cell = \_get\_rnn\_cell\_creator(cell\_type)

cells = [Cell(output\_size) for output\_size in num\_units]

cells = \_apply\_dropout\_wrapper(cells, dropout)

multi\_cell = tf.nn.rnn\_cell.MultiRNNCell(cells)

def stacked\_rnn\_cell(inputs, sequence\_lengths):

with tf.variable\_scope(scope\_name):

outputs, final\_states = tf.nn.static\_rnn(

multi\_cell,

tf.unstack(inputs, axis=1),

dtype=inputs.dtype,

sequence\_length=sequence\_lengths)

return final\_states[-1].h

return stacked\_rnn\_cell

def \_create\_regular\_rnn\_cell(num\_units, dropout, cell\_type, is\_bidirectional):

if is\_bidirectional:

return \_create\_bidirectional\_rnn\_cell(num\_units, dropout, cell\_type)

else:

return \_create\_unidirectional\_rnn\_cell(num\_units, dropout, cell\_type)

class StackedRNN(twml.layers.Layer):

"""

Layer for stacking RNN modules.

This layer provides a unified interface for RNN modules that perform well on CPUs and GPUs.

Arguments:

num\_units:

A list specifying the number of units per layer.

dropout:

Dropout applied to the input of each cell.

If list, has to dropout used for each layer.

If number, the same amount of dropout is used everywhere.

Defaults to 0.

is\_training:

Flag to specify if the layer is used in training mode or not.

cell\_type:

Sepcifies the type of RNN. Can be "LSTM". "GRU" is not yet implemented.

is\_bidirectional:

Specifies if the stacked RNN layer is bidirectional.

This is for forward compatibility, this is not yet implemented.

Defaults to False.

"""

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

num\_units,

dropout=0,

is\_training=True,

cell\_type="LSTM",

is\_bidirectional=False,

name="stacked\_rnn"):

super(StackedRNN, self).\_\_init\_\_(name=name)

if (is\_bidirectional):

raise NotImplementedError("Bidirectional RNN is not yet implemented")

if (cell\_type != "LSTM"):

raise NotImplementedError("Only LSTMs are supported")

if not isinstance(num\_units, (list, tuple)):

num\_units = [num\_units]

else:

num\_units = num\_units

self.num\_layers = len(num\_units)

if not isinstance(dropout, (tuple, list)):

dropout = [dropout] \* self.num\_layers

else:

dropout = dropout

self.is\_training = is\_training

is\_gpu\_available = twml.contrib.utils.is\_gpu\_available()

same\_unit\_size = all(size == num\_units[0] for size in num\_units)

same\_dropout\_rate = any(val == dropout[0] for val in dropout)

self.stacked\_rnn\_cell = None

self.num\_units = num\_units

self.dropout = dropout

self.cell\_type = cell\_type

self.is\_bidirectional = is\_bidirectional

def build(self, input\_shape):

self.stacked\_rnn\_cell = \_create\_regular\_rnn\_cell(self.num\_units,

self.dropout,

self.cell\_type,

self.is\_bidirectional)

def call(self, inputs, sequence\_lengths):

"""

Arguments:

inputs:

A tensor of size [batch\_size, max\_sequence\_length, embedding\_size].

sequence\_lengths:

The length of each input sequence in the batch. Should be of size [batch\_size].

Returns:

final\_output

The output of at the end of sequence\_length.

"""

return self.stacked\_rnn\_cell(inputs, sequence\_lengths)

def stacked\_rnn(inputs, sequence\_lengths, num\_units,

dropout=0, is\_training=True,

cell\_type="LSTM", is\_bidirectional=False, name="stacked\_rnn"):

"""Functional interface for StackedRNN

Arguments:

inputs:

A tensor of size [batch\_size, max\_sequence\_length, embedding\_size].

sequence\_lengths:

The length of each input sequence in the batch. Should be of size [batch\_size].

num\_units:

A list specifying the number of units per layer.

dropout:

Dropout applied to the input of each cell.

If list, has to dropout used for each layer.

If number, the same amount of dropout is used everywhere.

Defaults to 0.

is\_training:

Flag to specify if the layer is used in training mode or not.

cell\_type:

Sepcifies the type of RNN. Can be "LSTM" or "GRU".

is\_bidirectional:

Specifies if the stacked RNN layer is bidirectional.

Defaults to False.

Returns

outputs, state.

"""

rnn = StackedRNN(num\_units, dropout, is\_training, cell\_type, is\_bidirectional, name)

return rnn(inputs, sequence\_lengths)