# pylint: disable=no-member,arguments-differ, attribute-defined-outside-init

"""

Implementing Full Dense Layer

"""

from twml.layers import Layer

import tensorflow.compat.v1 as tf

from tensorflow.python.layers import core

class FullDense(Layer):

"""

Full-connected, Dense input layer class.

This layer implements the operation:

.. code-block:: python

outputs = activation(inputs.weight + bias)

Where ``activation`` is the activation function passed as the ``activation``

argument (if not ``None``), ``weight`` is a weights matrix created by the layer,

and ``bias`` is a bias vector created by the layer.

However, this layer breaks up ``weight`` into ``num\_partitions`` parts,

for the purpose of even disribution of weights across parameter servers

for distributed training.

Note - This layer is created to allow distributed training optimizations,

but can also be used for single node training (e.g. hogwild) without

code modification

Arguments:

output\_size:

Integer or Long, dimensionality of the output space.

weight\_initializer:

Initializer function for the weight matrix.

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

weight\_constraint:

An optional projection function to be applied to the

weight after being updated by an `Optimizer` (e.g. used to implement

norm constraints or value constraints for layer weights). The function

must take as input the unprojected variable and must return the

projected variable (which must have the same shape). Constraints are

not safe to use when doing asynchronous distributed training.

bias\_constraint:

An optional projection function to be applied to the

bias after being updated by an `Optimizer`.

num\_partitions:

Number of pieces to partition the weights into. This layer does

column partitioning of the weights, which is equivalent to

processing the input tensor with multiple fully connected layers

of smaller output size, and then concatenating these outputs

activation:

Activation function (callable). Set it to None to maintain a linear activation.

use\_bias:

Boolean whether to include a bias parameter in the layer

bias\_initializer:

Initializer function for the bias.

bias\_regularizer:

Regularizer function for the bias.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

activity\_regularizer:

Regularizer function for the output.

trainable:

Boolean, if `True` also add variables to the graph collection

``GraphKeys.TRAINABLE\_VARIABLES`` (see `tf.Variable

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/Variable>`\_).

name:

String, the name of the layer. Layers with the same name will

share weights, but to avoid mistakes we require ``reuse=True`` in such cases.

Properties:

output\_size:

Python integer, dimensionality of the output space.

activation:

Activation function (callable).

weight\_initializer:

Initializer instance (or name) for the weight matrix.

bias\_initializer:

Initializer instance (or name) for the bias.

weights:

list of underlying weight and bias matrix components. no guarantee on order of elements

weight\_regularizer:

Regularizer instance for the weight matrix (callable)

bias\_regularizer:

Regularizer instance for the bias (callable).

activity\_regularizer:

Regularizer instance for the output (callable)

weight\_constraint:

Constraint function for the weight matrix.

bias\_constraint:

Constraint function for the bias.

"""

def \_\_init\_\_(self, output\_size,

weight\_initializer=None,

weight\_regularizer=None,

weight\_constraint=None,

bias\_constraint=None,

num\_partitions=3,

activation=None,

use\_bias=True,

bias\_initializer=tf.zeros\_initializer(),

bias\_regularizer=None,

activity\_regularizer=None,

trainable=True,

name=None,

\*\*kwargs):

super(FullDense, self).\_\_init\_\_(trainable=trainable, name=name, \*\*kwargs)

self.\_output\_sizes = self.\_get\_output\_partition\_sizes(output\_size, num\_partitions)

self.\_units = output\_size

self.\_activation = activation

self.\_weight\_initializer = weight\_initializer

self.\_bias\_initializer = bias\_initializer

self.\_weight\_regularizer = weight\_regularizer

self.\_bias\_regularizer = bias\_regularizer

self.\_weight\_constraint = weight\_constraint

self.\_bias\_constraint = bias\_constraint

self.\_use\_bias = use\_bias

# NOTE - many initializers depend on fan\_in and fan\_out

# - as such, initialization here may be different than

# - for a non-partitioned FullDense

self.\_parts = [core.Dense(units=out\_size,

activation=activation,

use\_bias=use\_bias,

kernel\_initializer=weight\_initializer,

bias\_initializer=bias\_initializer,

kernel\_regularizer=weight\_regularizer,

bias\_regularizer=bias\_regularizer,

activity\_regularizer=activity\_regularizer,

kernel\_constraint=weight\_constraint,

bias\_constraint=bias\_constraint,

trainable=trainable,

name=name,

\*\*kwargs) for out\_size in self.\_output\_sizes]

@staticmethod

def \_get\_output\_partition\_sizes(out\_size, num\_parts):

""" Returns the appropriate output sizes of the partitions """

boundaries = [out\_size \* n // num\_parts for n in range(num\_parts + 1)]

return [k - j for j, k in zip(boundaries[:], boundaries[1:])]

def build(self, input\_shapes):

""" Create the appropriately sized weights and biases in each layer partition """

if isinstance(input\_shapes, (list, tuple)):

input\_shape = input\_shapes[0]

is\_compatible = True

for other\_shape in input\_shapes[1:]:

is\_compatible &= input\_shape.is\_compatible\_with(other\_shape)

if not is\_compatible:

raise ValueError("Input shapes %s are not compatible." % input\_shapes)

else:

input\_shape = input\_shapes

for part in self.\_parts:

part.build(input\_shape)

self.built = True

@property

def units(self):

""" Returns the number of output units of the layer """

return self.\_units

@property

def output\_size(self):

""" Returns the number of output units of the layer """

return self.\_units

@property

def activation(self):

""" Returns the activation function """

return self.\_activation

@property

def weight\_initializer(self):

""" Returns the weight\_initializer """

return self.\_weight\_initializer

@property

def weight\_regularizer(self):

""" Returns the weight\_regularizer """

return self.\_weight\_regularizer

@property

def weight\_constraint(self):

""" Returns the weight\_constraint """

return self.\_weight\_constraint

@property

def bias\_initializer(self):

""" Returns the bias\_initializer """

return self.\_bias\_initializer

@property

def bias\_regularizer(self):

""" Returns the bias\_regularizer """

return self.\_bias\_regularizer

@property

def bias\_constraint(self):

""" Returns the bias\_constraint """

return self.\_bias\_constraint

@property

def use\_bias(self):

""" Returns whether a bias is used in the layer """

return self.\_use\_bias

@property

def trainable\_variables(self):

""" Returns the trainable variables of the layer """

trainable\_vars = []

for pt in self.\_parts:

trainable\_vars += pt.trainable\_variables

return trainable\_vars

@property

def trainable\_weights(self):

""" Returns the trainable variables of the layer """

return self.trainable\_variables

@property

def non\_trainable\_variables(self):

""" Returns the non-trainable variables of the layer """

non\_trainable\_vars = []

for pt in self.\_parts:

non\_trainable\_vars += pt.non\_trainable\_variables

return non\_trainable\_vars

@property

def non\_trainable\_weights(self):

""" Returns the non-trainable variables of the layer """

return self.non\_trainable\_variables

@property

def variables(self):

""" Returns a list of all weights and biases in this layer """

layer\_vars = []

for pt in self.\_parts:

layer\_vars += pt.weights

return layer\_vars

@property

def weights(self):

""" Returns a list of all weights and biases in this layer """

return self.variables

@property

def dtype(self):

""" Returns the dtype of the layers weights """

return self.\_parts[0].dtype

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

"""The logic of the layer lives here.

Arguments:

inputs:

A dense Tensor or a list of such.

If `inputs` is a list, all tensors must have same `dense\_shape`.

Returns:

- If `inputs` is `SparseTensor`, then returns `bias + inputs \* dense\_b`.

- If `inputs` is a `list[SparseTensor`, then returns

`bias + accumulate\_n([sp\_a \* dense\_b for sp\_a in inputs])`.

"""

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

inputs = [inputs]

outputs = []

for inp in inputs:

part\_outputs = [part(inp) for part in self.\_parts]

outputs.append(tf.concat(part\_outputs, axis=-1))

return tf.accumulate\_n(outputs)

def full\_dense(inputs, output\_size,

weight\_initializer=None,

weight\_regularizer=None,

weight\_constraint=None,

bias\_constraint=None,

num\_partitions=3,

activation=None,

use\_bias=True,

bias\_initializer=tf.zeros\_initializer(),

bias\_regularizer=None,

activity\_regularizer=None,

trainable=True,

name=None,

reuse=None,

\*\*kwargs):

"""Functional interface for the fully-connected dense-input layer.

This layer implements the operation:

`outputs = activation(inputs.weight + bias)`

Where `activation` is the activation function passed as the `activation`

argument (if not `None`), `weight` is a weights matrix created by the layer,

and `bias` is a bias vector created by the layer

(only if `use\_bias` is `True`).

However, this layer breaks up ``weight`` into ``num\_partitions`` parts,

for the purpose of even disribution of weights across parameter servers

for distributed training.

Note - This layer is created to allow distributed training optimizations,

but can also be used for single node training (e.g. hogwild) without

code modification

Arguments:

inputs: Tensor input.

output\_size: Integer or Long, dimensionality of the output space.

weight\_initializer: Initializer function for the weight matrix.

If `None` (default), weights are initialized using the default

initializer used by `tf.get\_variable`.

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

weight\_constraint:

An optional projection function to be applied to the

weight after being updated by an `Optimizer` (e.g. used to implement

norm constraints or value constraints for layer weights). The function

must take as input the unprojected variable and must return the

projected variable (which must have the same shape). Constraints are

not safe to use when doing asynchronous distributed training.

bias\_constraint:

An optional projection function to be applied to the

bias after being updated by an `Optimizer`.

num\_partitions:

Number of pieces to partition the weights into. This layer does

column partitioning of the weights, which is equivalent to

processing the input tensor with multiple fully connected layers

of smaller output size, and then concatenating these outputs

activation: Activation function (callable). Set it to None to maintain a

linear activation.

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

bias\_initializer:

Initializer function for the bias.

bias\_regularizer:

Regularizer function for the bias.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

activity\_regularizer:

Regularizer function for the output.

trainable:

Boolean, if `True` also add variables to the graph collection

`GraphKeys.TRAINABLE\_VARIABLES` (see `tf.Variable`).

name:

String, the name of the layer.

reuse:

Boolean, whether to reuse the weights of a previous layer

by the same name.

Returns:

Output tensor with shape `inputs.shape[:-1] + [output\_size]`.

"""

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

inputs = [inputs]

dtype = inputs[0].dtype.base\_dtype

layer = FullDense(output\_size=output\_size,

weight\_initializer=weight\_initializer,

weight\_regularizer=weight\_regularizer,

weight\_constraint=weight\_constraint,

bias\_constraint=bias\_constraint,

num\_partitions=num\_partitions,

activation=activation,

use\_bias=use\_bias,

bias\_initializer=bias\_initializer,

bias\_regularizer=bias\_regularizer,

activity\_regularizer=activity\_regularizer,

trainable=trainable,

name=name,

dtype=dtype,

\_scope=name,

\_reuse=reuse,

\*\*kwargs)

return layer(inputs)