import numpy as np

from tensorflow.keras import backend as K

class VarianceScaling(object):

"""Initializer capable of adapting its scale to the shape of weights.

With `distribution="normal"`, samples are drawn from a truncated normal

distribution centered on zero, with `stddev = sqrt(scale / n)` where n is:

- number of input units in the weight tensor, if mode = "fan\_in"

- number of output units, if mode = "fan\_out"

- average of the numbers of input and output units, if mode = "fan\_avg"

With `distribution="uniform"`,

samples are drawn from a uniform distribution

within [-limit, limit], with `limit = sqrt(3 \* scale / n)`.

# Arguments

scale: Scaling factor (positive float).

mode: One of "fan\_in", "fan\_out", "fan\_avg".

distribution: Random distribution to use. One of "normal", "uniform".

seed: A Python integer. Used to seed the random generator.

# Raises

ValueError: In case of an invalid value for the "scale", mode" or

"distribution" arguments."""

def \_\_init\_\_(

self,

scale=1.0,

mode="fan\_in",

distribution="normal",

seed=None,

fan\_in=None,

fan\_out=None,

):

self.fan\_in = fan\_in

self.fan\_out = fan\_out

if scale <= 0.0:

raise ValueError("`scale` must be a positive float. Got:", scale)

mode = mode.lower()

if mode not in {"fan\_in", "fan\_out", "fan\_avg"}:

raise ValueError(

"Invalid `mode` argument: " 'expected on of {"fan\_in", "fan\_out", "fan\_avg"} ' "but got",

mode,

)

distribution = distribution.lower()

if distribution not in {"normal", "uniform"}:

raise ValueError(

"Invalid `distribution` argument: " 'expected one of {"normal", "uniform"} ' "but got",

distribution,

)

self.scale = scale

self.mode = mode

self.distribution = distribution

self.seed = seed

def \_\_call\_\_(self, shape, dtype=None, partition\_info=None):

fan\_in = shape[-2] if self.fan\_in is None else self.fan\_in

fan\_out = shape[-1] if self.fan\_out is None else self.fan\_out

scale = self.scale

if self.mode == "fan\_in":

scale /= max(1.0, fan\_in)

elif self.mode == "fan\_out":

scale /= max(1.0, fan\_out)

else:

scale /= max(1.0, float(fan\_in + fan\_out) / 2)

if self.distribution == "normal":

stddev = np.sqrt(scale) / 0.87962566103423978

return K.truncated\_normal(shape, 0.0, stddev, dtype=dtype, seed=self.seed)

else:

limit = np.sqrt(3.0 \* scale)

return K.random\_uniform(shape, -limit, limit, dtype=dtype, seed=self.seed)

def get\_config(self):

return {

"scale": self.scale,

"mode": self.mode,

"distribution": self.distribution,

"seed": self.seed,

}

def customized\_glorot\_uniform(seed=None, fan\_in=None, fan\_out=None):

"""Glorot uniform initializer, also called Xavier uniform initializer.

It draws samples from a uniform distribution within [-limit, limit]

where `limit` is `sqrt(6 / (fan\_in + fan\_out))`

where `fan\_in` is the number of input units in the weight tensor

and `fan\_out` is the number of output units in the weight tensor.

# Arguments

seed: A Python integer. Used to seed the random generator.

# Returns

An initializer."""

return VarianceScaling(

scale=1.0,

mode="fan\_avg",

distribution="uniform",

seed=seed,

fan\_in=fan\_in,

fan\_out=fan\_out,

)

def customized\_glorot\_norm(seed=None, fan\_in=None, fan\_out=None):

"""Glorot norm initializer, also called Xavier uniform initializer.

It draws samples from a uniform distribution within [-limit, limit]

where `limit` is `sqrt(6 / (fan\_in + fan\_out))`

where `fan\_in` is the number of input units in the weight tensor

and `fan\_out` is the number of output units in the weight tensor.

# Arguments

seed: A Python integer. Used to seed the random generator.

# Returns

An initializer."""

return VarianceScaling(

scale=1.0,

mode="fan\_avg",

distribution="normal",

seed=seed,

fan\_in=fan\_in,

fan\_out=fan\_out,

)