"""

Provides a general optimizer for pruning features of a neural network.

The optimizer estimates the computational cost of features, combines this information with pruning

signals indicating their usefulness, and disables features via binary masks at regular intervals.

To make a layer prunable, use `twml.contrib.pruning.apply\_mask`:

dense1 = tf.layers.dense(inputs=inputs, units=50, activation=tf.nn.relu)

dense1 = apply\_mask(dense1)

To prune the network, apply PruningOptimizer to any cross-entropy loss:

loss = tf.losses.sparse\_softmax\_cross\_entropy(labels=labels, logits=logits)

optimizer = PruningOptimizer(learning\_rate=0.001, momentum=0.5)

minimize = optimizer.minimize(

loss=loss,

prune\_every=10,

burn\_in=100,

global\_step=tf.train.get\_global\_step())

"""

import tensorflow.compat.v1 as tf

from twml.contrib.pruning import computational\_cost, prune, update\_pruning\_signals

from twml.contrib.pruning import MASK\_COLLECTION

class PruningOptimizer(tf.train.MomentumOptimizer):

"""

Updates parameters with SGD and pruning masks using Fisher pruning.

Arguments:

learning\_rate: float

Learning rate of SGD

momentum: float

Momentum used by SGD

use\_locking: bool

If `True`, use locks for update operations

name: str

Optional name prefix for the operations created when applying gradients

use\_nesterov: bool

If `True`, use Nesterov momentum

"""

def \_\_init\_\_(

self,

learning\_rate,

momentum=0.9,

use\_locking=False,

name="PruningOptimizer",

use\_nesterov=False):

super(PruningOptimizer, self).\_\_init\_\_(

learning\_rate=learning\_rate,

momentum=momentum,

use\_locking=use\_locking,

name=name,

use\_nesterov=use\_nesterov)

def minimize(

self,

loss,

prune\_every=100,

burn\_in=0,

decay=.96,

flops\_weight='AUTO',

flops\_target=0,

update\_params=None,

method='Fisher',

\*args,

\*\*kwargs):

"""

Create operations to minimize loss and to prune features.

A pruning signal measures the importance of feature maps. This is weighed against the

computational cost of computing a feature map. Features are then iteratively pruned

based on a weighted average of feature importance S and computational cost C (in FLOPs):

$$S + w \* C$$

Setting `flops\_weight` to 'AUTO' is the most convenient and recommended option, but not

necessarily optimal.

Arguments:

loss: tf.Tensor

The value to minimize

prune\_every: int

One entry of a mask is set to zero only every few update steps

burn\_in: int

Pruning starts only after this many parameter updates

decay: float

Controls exponential moving average of pruning signals

flops\_weight: float or str

Controls the targeted trade-off between computational complexity and performance

flops\_target: float

Stop pruning when computational complexity is less or this many floating point ops

update\_params: tf.Operation

Optional training operation used instead of MomentumOptimizer to update parameters

method: str

Method used to compute pruning signal (currently only supports 'Fisher')

Returns:

A `tf.Operation` updating parameters and pruning masks

References:

\* Theis et al., Faster gaze prediction with dense networks and Fisher pruning, 2018

"""

# gradient-based updates of parameters

if update\_params is None:

update\_params = super(PruningOptimizer, self).minimize(loss, \*args, \*\*kwargs)

masks = tf.get\_collection(MASK\_COLLECTION)

with tf.variable\_scope('pruning\_opt', reuse=True):

# estimate computational cost per data point

batch\_size = tf.cast(tf.shape(masks[0].tensor), loss.dtype)[0]

cost = tf.divide(computational\_cost(loss), batch\_size, name='computational\_cost')

tf.summary.scalar('computational\_cost', cost)

if masks:

signals = update\_pruning\_signals(loss, masks=masks, decay=decay, method=method)

# estimate computational cost per feature map

costs = tf.gradients(cost, masks)

# trade off computational complexity and performance

if flops\_weight.upper() == 'AUTO':

signals = [s / (c + 1e-6) for s, c in zip(signals, costs)]

elif not isinstance(flops\_weight, float) or flops\_weight != 0.:

signals = [s - flops\_weight \* c for s, c in zip(signals, costs)]

counter = tf.Variable(0, name='pruning\_counter')

counter = tf.assign\_add(counter, 1, use\_locking=True)

# only prune every so often after a burn-in phase

pruning\_cond = tf.logical\_and(counter > burn\_in, tf.equal(counter % prune\_every, 0))

# stop pruning after reaching threshold

if flops\_target > 0:

pruning\_cond = tf.logical\_and(pruning\_cond, tf.greater(cost, flops\_target))

update\_masks = tf.cond(

pruning\_cond,

lambda: prune(signals, masks=masks),

lambda: tf.group(masks))

return tf.group([update\_params, update\_masks])

# no masks found

return update\_params