"""

This module implements tools for pruning neural networks.

In particular, it provides tools for dealing with masks:

features = apply\_mask(features)

The function `apply\_mask` applies a binary mask to the channels of a given tensor. Consider the

following loss:

logits = tf.matmul(features, weights)

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

Each mask has a corresponding pruning signal. The function `update\_pruning\_signals` will update and

return these signals:

signals = update\_pruning\_signals(loss)

The pruning operation will zero out the mask entry with the smallest corresponding pruning signal:

prune(signals)

The following function allows us to estimate the computational cost of a graph (number of FLOPs):

cost = computational\_cost(loss)

To compute the cost of each feature per data point, we can do:

costs = tf.gradients(cost / batch\_size, masks)

The current implementation of `computational\_cost` is designed to work with standard feed-forward

and convolutional network architectures only, but may fail with more complicated architectures.

"""

import numpy as np

import tensorflow.compat.v1 as tf

MASK\_COLLECTION = 'pruning/masks'

MASK\_EXTENDED\_COLLECTION = 'pruning/masks\_extended'

OP\_COLLECTION = 'pruning/ops'

def apply\_mask(tensor, name='pruning'):

"""

Point-wise multiplies a tensor with a binary mask.

During training, pruning is simulated by setting entries of the mask to zero.

Arguments:

tensor: tf.Tensor

A tensor where the last dimension represents channels which will be masked

Returns:

`tf.Tensor` with same shape as `tensor`

"""

tensor\_shape = tensor.shape

with tf.variable\_scope(name, reuse=True):

# allocate masks and corresponding pruning signals

mask = tf.Variable(tf.ones(tensor.shape.as\_list()[-1]), trainable=False, name='mask')

pruning\_signal = tf.Variable(tf.zeros\_like(mask), trainable=False, name='signal')

# extending masks is a trick to get a separate gradient for each data point

mask\_extended = extend\_mask(mask, tensor)

# store extended mask, pruning signal, and other vars for easy access later

mask.extended = mask\_extended

mask.pruning\_signal = pruning\_signal

mask.tensor = tensor

# mask tensor

tensor = tf.multiply(tensor, mask\_extended)

tensor.set\_shape(tensor\_shape)

tensor.\_mask = mask

tf.add\_to\_collection(MASK\_COLLECTION, mask)

tf.add\_to\_collection(MASK\_EXTENDED\_COLLECTION, mask.extended)

tf.add\_to\_collection(OP\_COLLECTION, tensor.op)

return tensor

def extend\_mask(mask, tensor):

"""

Repeats the mask for each data point stored in a tensor.

If `tensor` is AxBxC dimensional and `mask` is C dimensional, returns an Ax1xC dimensional

tensor with A copies or `mask`.

Arguments:

mask: tf.Tensor

The mask which will be extended

tensor: tf.Tensor

The tensor to which the extended mask will be applied

Returns:

The extended mask

"""

batch\_size = tf.shape(tensor)[:1]

ones = tf.ones([tf.rank(tensor) - 1], dtype=batch\_size.dtype)

multiples = tf.concat([batch\_size, ones], 0)

mask\_shape = tf.concat([ones, [-1]], 0)

return tf.tile(tf.reshape(mask, mask\_shape), multiples)

def find\_input\_mask(tensor):

"""

Find ancestral mask affecting the number of pruned channels of a tensor.

Arguments:

tensor: tf.Tensor

Tensor for which to identify relevant mask

Returns:

A `tf.Tensor` or `None`

"""

if hasattr(tensor, '\_mask'):

return tensor.\_mask

if tensor.op.type in ['MatMul', 'Conv1D', 'Conv2D', 'Conv3D', 'Transpose']:

# op produces a new number of channels, preceding mask therefore irrelevant

return None

if not tensor.op.inputs:

return None

for input in tensor.op.inputs:

mask = find\_input\_mask(input)

if mask is not None:

return mask

def find\_output\_mask(tensor):

"""

Find mask applied to the tensor or one of its descendants if it affects the tensor's pruned shape.

Arguments:

tensor: tf.Tensor or tf.Variable

Tensor for which to identify relevant mask

Returns:

A `tf.Tensor` or `None`

"""

if isinstance(tensor, tf.Variable):

return find\_output\_mask(tensor.op.outputs[0])

if hasattr(tensor, '\_mask'):

return tensor.\_mask

for op in tensor.consumers():

if len(op.outputs) != 1:

continue

if op.type in ['MatMul', 'Conv1D', 'Conv2D', 'Conv3D']:

# masks of descendants are only relevant if tensor is right-multiplied

if tensor == op.inputs[1]:

return find\_output\_mask(op.outputs[0])

return None

mask = find\_output\_mask(op.outputs[0])

if mask is not None:

return mask

def find\_mask(tensor):

"""

Returns masks indicating channels of the tensor that are effectively removed from the graph.

Arguments:

tensor: tf.Tensor

Tensor for which to compute a mask

Returns:

A `tf.Tensor` with binary entries indicating disabled channels

"""

input\_mask = find\_input\_mask(tensor)

output\_mask = find\_output\_mask(tensor)

if input\_mask is None:

return output\_mask

if output\_mask is None:

return input\_mask

if input\_mask is output\_mask:

return input\_mask

return input\_mask \* output\_mask

def pruned\_shape(tensor):

"""

Computes the shape of a tensor after taking into account pruning of channels.

Note that the shape will only differ in the last dimension, even if other dimensions are also

effectively disabled by pruning masks.

Arguments:

tensor: tf.Tensor

Tensor for which to compute a pruned shape

Returns:

A `tf.Tensor[tf.float32]` representing the pruned shape

"""

mask = find\_mask(tensor)

if mask is None:

return tf.cast(tf.shape(tensor), tf.float32)

return tf.concat([

tf.cast(tf.shape(tensor)[:-1], mask.dtype),

tf.reduce\_sum(mask, keepdims=True)], 0)

def computational\_cost(op\_or\_tensor, \_observed=None):

"""

Estimates the computational complexity of a pruned graph (number of floating point operations).

This function currently only supports sequential graphs such as those of MLPs and

simple CNNs with 2D convolutions in NHWC format.

Note that the computational cost returned by this function is proportional to batch size.

Arguments:

op\_or\_tensor: tf.Tensor or tf.Operation

Root node of graph for which to compute computational cost

Returns:

A `tf.Tensor` representing a number of floating point operations

"""

cost = tf.constant(0.)

# exclude cost of computing extended pruning masks

masks\_extended = [mask.extended for mask in tf.get\_collection(MASK\_COLLECTION)]

if op\_or\_tensor in masks\_extended:

return cost

# convert tensor to op

op = op\_or\_tensor.op if isinstance(op\_or\_tensor, (tf.Tensor, tf.Variable)) else op\_or\_tensor

# make sure cost of op will not be counted twice

if \_observed is None:

\_observed = []

elif op in \_observed:

return cost

\_observed.append(op)

# compute cost of computing inputs

for tensor in op.inputs:

cost = cost + computational\_cost(tensor, \_observed)

# add cost of operation

if op.op\_def is None or op in tf.get\_collection(OP\_COLLECTION):

# exclude cost of undefined ops and pruning ops

return cost

elif op.op\_def.name == 'MatMul':

shape\_a = pruned\_shape(op.inputs[0])

shape\_b = pruned\_shape(op.inputs[1])

return cost + shape\_a[0] \* shape\_b[1] \* (2. \* shape\_a[1] - 1.)

elif op.op\_def.name in ['Add', 'Mul', 'BiasAdd']:

return cost + tf.cond(

tf.size(op.inputs[0]) > tf.size(op.inputs[1]),

lambda: tf.reduce\_prod(pruned\_shape(op.inputs[0])),

lambda: tf.reduce\_prod(pruned\_shape(op.inputs[1])))

elif op.op\_def.name in ['Conv2D']:

output\_shape = pruned\_shape(op.outputs[0])

input\_shape = pruned\_shape(op.inputs[0])

kernel\_shape = pruned\_shape(op.inputs[1])

inner\_prod\_cost = (tf.reduce\_prod(kernel\_shape[:2]) \* input\_shape[-1] \* 2. - 1.)

return cost + tf.reduce\_prod(output\_shape) \* inner\_prod\_cost

return cost

def update\_pruning\_signals(loss, decay=.96, masks=None, method='Fisher'):

"""

For each mask, computes corresponding pruning signals indicating the importance of a feature.

Arguments:

loss: tf.Tensor

Any cross-entropy loss

decay: float

Controls exponential moving average of pruning signals

method: str

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

Returns:

A `list[tf.Tensor]` of pruning signals corresponding to masks

References:

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

"""

if masks is None:

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

if method not in ['Fisher']:

raise ValueError('Pruning method \'{0}\' not supported.'.format(method))

if not masks:

return []

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

# compute gradients of extended masks (yields separate gradient for each data point)

grads = tf.gradients(loss, [m.extended for m in masks])

# estimate Fisher pruning signals from batch

signals\_batch = [tf.squeeze(tf.reduce\_mean(tf.square(g), 0)) for g in grads]

# update pruning signals

signals = [m.pruning\_signal for m in masks]

signals = [tf.assign(s, decay \* s + (1. - decay) \* f, use\_locking=True)

for s, f in zip(signals, signals\_batch)]

return signals

def prune(signals, masks=None):

"""

Prunes a single feature by zeroing the mask entry with the smallest pruning signal.

Arguments:

signals: list[tf.Tensor]

A list of pruning signals

masks: list[tf.Tensor]

A list of corresponding masks, defaults to `tf.get\_collection(MASK\_COLLECTION)`

Returns:

A `tf.Operation` which updates masks

"""

if masks is None:

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

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

# make sure we don't select already pruned units

signals = [tf.where(m > .5, s, tf.zeros\_like(s) + np.inf) for m, s in zip(masks, signals)]

# find units with smallest pruning signal in each layer

min\_idx = [tf.argmin(s) for s in signals]

min\_signals = [s[i] for s, i in zip(signals, min\_idx)]

# find layer with smallest pruning signal

l = tf.argmin(min\_signals)

# construct pruning operations, one for each mask

updates = []

for k, i in enumerate(min\_idx):

# set mask of layer l to 0 where pruning signal is smallest

updates.append(

tf.cond(

tf.equal(l, k),

lambda: tf.scatter\_update(

masks[k], tf.Print(i, [i], message="Pruning layer [{0}] at index ".format(k)), 0.),

lambda: masks[k]))

updates = tf.group(updates, name='prune')

return updates