```
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_hub as hub
import tensorflow_probability as tfp
import skimage.io as io
from skimage.filters import threshold_otsu
from skimage.measure import label, regionprops
from skimage.morphology import closing, square
from collections import defaultdict
import pickle
import os
import glob
import blur


# From:
# https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only allocate 1GB of memory on the first GPU
    try:
        tf.config.experimental.set_virtual_device_configuration(
            gpus[0],
            [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=3072)
])
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)


# Constants
IMG_SIZE = [ 224, 224 ]
kVal = 5    # Take top-k predictions
alpha = 0.5

# Loss function
bce = tf.keras.losses.BinaryCrossentropy( from_logits=True )

# Batching
TOTAL_IMAGES = 500
BATCH_SIZE = 25
NUM_BATCHES = TOTAL_IMAGES//BATCH_SIZE
```

```
# Ensure number of images is correct
assert( NUM_BATCHES * BATCH_SIZE == TOTAL_IMAGES )

# IMG_DIMS is [ None, IMG_SIZE, 3 ]
IMG_DIMS = [ None ]
IMG_DIMS.extend( IMG_SIZE )
IMG_DIMS.extend( [3] )

# TFRecords
recPath = 'records'             # Location of TFRecords
recName = 'ImageNet'
recSize = 1500                  # Number of examples per TFRecord
recNum = 2                      # Which TFRecord to load from
offset = recSize * (recNum-1)   # Offset for the ground truth labels


# Read TFRecord file from:
# https://stackoverflow.com/questions/47861084/how-to-store-numpy-arrays-as-
tfrecord
def _parse_tfr_element(element):

    parse_dic = {
            'image': tf.io.FixedLenFeature([], tf.string), # Note that it is tf.s
tring, not tf.float32
            'label': tf.io.FixedLenFeature([], tf.string),
            'bbox': tf.io.FixedLenFeature([], tf.string),
    }
    example_message = tf.io.parse_single_example(element, parse_dic)

    b_image = example_message['image'] # get byte string
    b_bbox = example_message['bbox']
    b_label = example_message['label']

    img = tf.io.parse_tensor(b_image, out_type=tf.uint8) # restore 2D array from
byte string
    bbox = tf.io.parse_tensor(b_bbox, out_type=tf.int32)
    label = tf.io.parse_tensor(b_label, out_type=tf.string)
    label = int(label)

    return img, label, bbox


def normalize_img( image, label, bbox ):
    """Normalizes images: `uint8` -> `float32`."""
```

```
    return tf.cast(image, tf.float32) / 255, label, bbox


# Python function to manipulate dataset
def map_func( image, label, bbox ):
    """ Scales images to IMG_SIZE.
        Removes bounding box element of dataset."""

    # Deal with grayscale images
    if len( tf.shape(image) ) == 2:
        image = np.expand_dims( image, axis=-1 )
        image = tf.concat( [image, image, image], axis=-1 )

    image = tf.image.resize( image, IMG_SIZE )

    # # Can optionally apply foveation here
    # image = blur.applyBlur( image, IMG_SIZE[0]//2, IMG_SIZE[1]//2, 70 )

    return image, label


# Function to define shape of tfds
def ensureShape( image, label ):

    # dims -> [ IMG_SIZE, 3 ]
    dims = []
    dims.extend( IMG_SIZE )
    dims.extend( [3] )

    image = tf.ensure_shape( image, dims )

    return image, label


def calcAcc( probs, truth, k ):

    numEx = tf.shape( probs )[0]

    correctBools = tf.math.in_top_k( truth[ np.arange( offset,offset+numEx ) ], p
robs, kVal )
    numCorrect = tf.math.reduce_sum( tf.cast( correctBools, tf.float32 ) )
    print( numCorrect )
    print( numCorrect / tf.cast( numEx, tf.float32 ) )

    return
```

```
def sortRecs( rec ):

    fileName, _ = rec.split( '.' )
    _, num = fileName.split( '-' )
    return int(num)


if __name__ == "__main__":

    # Load data
    fileName = os.path.join(recPath, recName + '-' + str(recNum) + '.tfrecords')

    # # Iterate through all images of a specific extension in the specified direc
tory
    # fileName = []
    # imgPath = os.path.join( recPath, '*.tfrecords' )

    # for filepath in glob.iglob( imgPath ):
    #     #print(filepath)
    #     fileName.append( filepath )

    # # Sort list of tfrecords in numerical ascending order b/c ground truth labe
ls are in that order
    # fileName.sort( key=sortRecs )
    # print( fileName )



    tfr_dataset = tf.data.TFRecordDataset(fileName)
    dataset = tfr_dataset.map(_parse_tfr_element)

    print("\n\n")
    print( dataset.element_spec )

    # Map dataset
    ds = dataset.map(
        normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE)
    print( ds.element_spec )

    # Map using tf.py_function
    ds = ds.map( lambda image, label, bbox: tf.py_function(func=map_func,
          inp=[image, label, bbox], Tout=[tf.float32, tf.int32]),
          num_parallel_calls=tf.data.experimental.AUTOTUNE )
```

```
    # Set (previously known) shapes of images
    ds = ds.map(
        ensureShape, num_parallel_calls=tf.data.experimental.AUTOTUNE)
    ds = ds.batch( BATCH_SIZE )

    print( ds.element_spec )

    # Load mapped ground truth labels from a file
    with open('truthMapped.pkl', 'rb') as f:
        data = f.read()
        mappedTruthDict = pickle.loads(data)

    # Use mappings to get the correct labels
    mappedTruthDict = { k:v[0] for (k,v) in mappedTruthDict.items() }
    truth = np.array( list( mappedTruthDict.values() ) )

    # Load pre-trained model
    model = tf.keras.Sequential([
        hub.KerasLayer("https://tfhub.dev/google/imagenet/inception_v1/classifica
tion/4"),
    ])
    model.build( IMG_DIMS )  # Batch input shape
    model.summary()

    # Store the labels
    topKOri = np.zeros( [BATCH_SIZE, kVal] )
    topKSq = np.zeros( [BATCH_SIZE, kVal, kVal] )
    confs = np.zeros( [BATCH_SIZE, kVal, kVal] )

    numCorr = 0
    count = 0

    for I, tempLabel in ds.take( NUM_BATCHES ).cache().prefetch(tf.data.experimen
tal.AUTOTUNE):

        # img = I

        img = []
        for i in range( BATCH_SIZE ):
            img.append( blur.applyBlur( I[i], 112, 112, 70 ) )
        img = tf.convert_to_tensor( img )

        # Ensure the shape is correct
        img = tf.reshape( img, [BATCH_SIZE, 224, 224, 3] )
```

```
        imageVar = tf.Variable( img )

        for topLabel in range(kVal):

            with tf.GradientTape() as tape:

                # Watch the input image to compute saliency map later
                tape.watch( imageVar )

                # Forward-pass to get initial predictions
                probs = model( imageVar )

                # Get top-k predictions
                logits, preds = tf.math.top_k(probs, k=kVal)    # Throw out the p
robs for each top prediction (included in probs variable)

                true = tf.one_hot( preds, len( probs[0] )  )    # One-
hot encode the predictions to the same size as probs
                loss = bce( probs, true[:,topLabel,:] )


            grads = abs( tape.gradient( loss, imageVar ) )
            grads = tf.reduce_max( grads, axis=-1 )

            # Save original predictions
            dictOri = []
            for i in range( BATCH_SIZE ):
                dictOri.append( dict( zip( preds[i].numpy(), tf.nn.softmax( logit
s[i] ).numpy() ) ) )    # Apply a softmax to normalize for comparison

            for i in range( BATCH_SIZE ):

                # Find 80th percentile and apply first threshold
                thres = tfp.stats.percentile( grads[i], q=80 )
                image = tf.keras.activations.relu( grads[i], threshold=thres ).nu
mpy()


                # Apply threshold
                thres = threshold_otsu(image)
                bw = closing(image > thres, square(3))

                # label image regions
                label_image = label(bw)

                # Get max region
```

```
                curMaxArea = 0
                for region in regionprops(label_image):

                    if region.area >= curMaxArea:

                        curMaxArea = region.area
                        maxRegion = region

                minr, minc, maxr, maxc = maxRegion.bbox

                # Second pass
                height = maxc - minc
                width = maxr - minr
                f_new = np.floor( alpha * np.max( [height, width] ) )
                f_new = np.max( [30, f_new] )   # Minimum foveal size
                centerX = minc + height/2
                centerY = minr + width/2

                imgSec = blur.applyBlur( I[i], centerX, centerY, f_new )
                # imgSec = blur.applyBlur( I[i], centerX, centerY, 70 )

                imgSec = tf.reshape( imgSec, [1, 224, 224, 3] )

                logits = model.predict( imgSec )
                conf, preds = tf.math.top_k(logits, k=kVal)
                confs[i][ topLabel ] = conf
                topKSq[i][ topLabel ] = preds

        # Map top-k into dicts
        dictsBatched = []
        for i in range( BATCH_SIZE ):
            dicts = []
            for j in range( kVal ):
                dicts.append( dict( zip( topKSq[i][j], tf.nn.softmax( confs[i][j]
).numpy() ) ) )     # Apply a softmax to normalize for comparison
            dictsBatched.append( dicts )

        # Get the highest confidences for each unique label
        dictTopK = defaultdict(int)

        # Get predictions for each example in the batch
        for i in range( BATCH_SIZE ):

            # Reset
            dictTopK.clear()
```

```
            dicts = dictsBatched[i]

            # Do not include the original top-k
            dictTopK.update( dicts[0] )
            for j in range(1,kVal):
                dictTopK.update( (k,v) for k,v in dicts[j].items() if dictTopK[k]
 < v )

            # Include the original top-k
            # dictTopK.update( dictOri[i] )
            # for i in range(kVal):
            #     dictTopK.update( (k,v) for k,v in dicts[i].items() if dictTopK[
k] < v )

            # Sort the dict in descending order
            # Get topK labels
            tupleTopK = sorted(dictTopK.items(), key=lambda x: x[1], reverse=True
)[:kVal]

            # Get labels into a list
            newTopK = [ int(x[0]) for x in tupleTopK ]

            # Get true label from ground truth
            trueLabel = truth[ offset+count ]
            count += 1

            if trueLabel in newTopK:
                numCorr += 1
            else:
                print( newTopK )
                print( trueLabel )
                print( "Completed", count )

    print( numCorr, TOTAL_IMAGES )
    print( "Percent Correct:", numCorr/TOTAL_IMAGES )
    print( "Alpha:", alpha )
```