Derek Lee Deep Learning Fall 2020

Professor Curro Midterm

**Adaptive Label Smoothing**

import numpy as np

import tensorflow as tf

import tensorflow\_datasets as tfds

import matplotlib.pyplot as plt

import os

from pathlib import Path

from xml.etree import ElementTree

import itertools

import gc

# Use masking

# # # SET TO FALSE # # #

# # # INCORRECT USAGE # # #

MASKING = False

# Use Adaptive Label Smoothing

    # 0 = Hard Label

    # 1 = Full ALS

ALS = False

BETA = 0

# Train or Test

train = False

test = True

# 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=4096)])

        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)

annotations\_dir\_path = os.path.join( os.getcwd(), "annotations" )

xmls\_dir\_path = os.path.join( annotations\_dir\_path, "xmls" )

unmarshallers = {

    "xmin": lambda x: int(x.text),

    "ymin": lambda x: int(x.text),

    "xmax": lambda x: int(x.text),

    "ymax": lambda x: int(x.text),

    "width": lambda x: int(x.text),

    "height": lambda x: int(x.text)

}

# Ignore the following training examples

# Missing bounding boxes

missingXMLs = {

    'samoyed\_10.xml',

    'saint\_bernard\_15.xml',

    'Abyssinian\_104.xml',

    'Ragdoll\_199.xml',

    'Bengal\_175.xml',

    'Bengal\_111.xml',

    'Bengal\_105.xml'    # Has multiple bounding boxes

}

# Constants

NUM\_EPOCHS = 200

BATCH\_SIZE = 16

LEARN\_RATE = 0.001

IMG\_SIZE = 100

# Training

NUM\_CLASSES = 37

OPTIMIZER = tf.keras.optimizers.SGD( learning\_rate = LEARN\_RATE, momentum = 0.9 )

REGULARIZER = tf.keras.regularizers.l2( 0.01 )

LOSS = tf.keras.losses.CategoricalCrossentropy( from\_logits = False )

METRIC = tf.keras.metrics.CategoricalAccuracy()

checkpoint\_path = os.getcwd() + "training\_1/cp-{epoch:04d}.ckpt"

checkpoint\_dir = os.path.dirname(checkpoint\_path)

# Create a callback that saves the model's weights from:

# https://www.tensorflow.org/tutorials/keras/save\_and\_load

cp\_callback = tf.keras.callbacks.ModelCheckpoint(

    filepath=checkpoint\_path,

    verbose=1,

    save\_weights\_only=True,

    period=20)  # Saves every 10 epochs

# Decreases learning rate at specific epochs

def lrdecay(epoch):

    lr = LEARN\_RATE

    if epoch > 150:

        lr \*= 1e-3

    elif epoch > 100:

        lr \*= 1e-2

    elif epoch > 50:

        lr \*= 1e-1

    return lr

# Display some plots

def testLoad( data, info ):

    plt.figure( figsize=(10,10) )

    i=0

    for image, label in data:

        if i == 25:

            break

        plt.subplot( 5, 5, i+1 )

        plt.xticks([])

        plt.yticks([])

        plt.grid( False )

        plt.imshow( image )

        label = info.features["label"].int2str(label)

        plt.xlabel( label )

        i += 1

    plt.show()

# Architecture based on Full Pre-Activation from:

# https://arxiv.org/pdf/1603.05027.pdf

# Implementation inspired by:

# https://www.tensorflow.org/tutorials/customization/custom\_layers

class identityBlock( tf.keras.Model ):

    def \_\_init\_\_( self, filters ):

        super( identityBlock, self ).\_\_init\_\_( name = '' )

        f1, f2 = filters

        k = 3   # Kernel size

        self.conv2a = tf.keras.layers.Conv2D( f1, kernel\_size = (1, 1), strides = (1, 1), padding = 'valid', kernel\_regularizer = REGULARIZER )

        self.bn2a = tf.keras.layers.BatchNormalization()

        self.conv2b = tf.keras.layers.Conv2D( f1, kernel\_size = (k, k), strides = (1, 1), padding = 'same', kernel\_regularizer = REGULARIZER )

        self.bn2b = tf.keras.layers.BatchNormalization()

        self.conv2c = tf.keras.layers.Conv2D( f2, kernel\_size = (1, 1), strides = (1, 1), padding = 'valid', kernel\_regularizer = REGULARIZER )

        self.bn2c = tf.keras.layers.BatchNormalization()

    def call( self, inputTensor, training = False ):

                x = inputTensor

        # Block 1

        x = self.bn2a( x, training = training )

        x = tf.nn.leaky\_relu( x )

        x = self.conv2a( x )

        # Block 2

        x = self.bn2b( x, training = training )

        x = tf.nn.leaky\_relu( x )

        x = self.conv2b( x )

        # Block 3

        x = self.bn2c( x, training = training )

        x = tf.nn.relu( x )

        x = self.conv2c( x )

        # Output

        x += inputTensor

        return x

class convBlock( tf.keras.Model ):

    def \_\_init\_\_( self, filters, s ):

        super( convBlock, self ).\_\_init\_\_( name = '' )

        f1, f2 = filters

        k = 3   # Kernel size

        self.conv2a = tf.keras.layers.Conv2D( f1, kernel\_size = (1, 1), strides = (s, s), padding = 'valid', kernel\_regularizer = REGULARIZER )

        self.bn2a = tf.keras.layers.BatchNormalization()

        self.conv2b = tf.keras.layers.Conv2D( f1, kernel\_size = (k, k), strides = (1, 1), padding = 'same', kernel\_regularizer = REGULARIZER )

        self.bn2b = tf.keras.layers.BatchNormalization()

        self.conv2c = tf.keras.layers.Conv2D( f2, kernel\_size = (1, 1), strides = (1, 1), padding = 'valid', kernel\_regularizer = REGULARIZER )

        self.bn2c = tf.keras.layers.BatchNormalization()

        self.conv2Shortcut = tf.keras.layers.Conv2D( f2, kernel\_size = (1, 1), strides = (s, s), padding = 'valid', kernel\_regularizer = REGULARIZER )

        self.bn2Shortcut = tf.keras.layers.BatchNormalization()

    def call( self, inputTensor, training = False ):

        x = inputTensor

        xShort = inputTensor

        # Block 1

        x = self.conv2a( x )

        # Block 2

        x = self.bn2b( x, training = training )

        x = tf.nn.leaky\_relu( x )

        x = self.conv2b( x )

        # Block 3

        x = self.bn2c( x, training = training )

        x = tf.nn.relu( x )

        x = self.conv2c( x )

        # Shortcut

        xShort = self.bn2Shortcut( xShort, training = training )

        xShort = tf.nn.relu( xShort )

        xShort = self.conv2Shortcut( xShort )

        # Output

        x += xShort

        return x

# Module containing Image Classification model

# Architecture based on:

# https://towardsdatascience.com/understand-and-implement-resnet-50-with-tensorflow-2-0-1190b9b52691

class imgClassMod( tf.Module ):

    def \_\_init\_\_( self ):

        self.model = tf.keras.models.Sequential()

        # Stem Layer

        self.model.add( tf.keras.layers.ZeroPadding2D( (3, 3) ) )

        self.model.add( tf.keras.layers.Conv2D( 64, (7, 7), strides = (2, 2) ) )

        self.model.add( tf.keras.layers.BatchNormalization() )

        self.model.add( tf.keras.layers.ReLU() )

        self.model.add( tf.keras.layers.MaxPooling2D( (3, 3), strides = (2, 2) ) )

        # Hidden Layers

        # Stage 1

        self.model.add( convBlock( filters = [ 64, 256 ], s = 1 ) )

        for \_ in range(2):

            self.model.add( identityBlock( filters = [ 64, 256 ] ) )

        # Stage 2

        self.model.add( convBlock( filters = [ 128, 512 ], s = 2 ) )

        for \_ in range(3):

            self.model.add( identityBlock( filters = [ 128, 512 ] ) )

        # Stage 3

        self.model.add( convBlock( filters = [ 256, 1024 ], s = 2 ) )

        for \_ in range(5):

            self.model.add( identityBlock( filters = [ 256, 1024 ] ) )

        # Stage 4

        self.model.add( convBlock( filters = [ 512, 2048 ], s = 2 ) )

        for \_ in range(2):

            self.model.add( identityBlock( filters = [ 512, 2048 ] ) )

        # Pooling

        self.model.add( tf.keras.layers.AveragePooling2D( (2, 2), padding = 'same' ) )

        # Output

        self.model.add( tf.keras.layers.Flatten() )

        self.model.add( tf.keras.layers.Dense( NUM\_CLASSES, activation = 'softmax' ) )

        self.lrdecay = tf.keras.callbacks.LearningRateScheduler(lrdecay)    # Learning rate decay

        self.model.compile( loss = LOSS,

                            optimizer = OPTIMIZER, metrics = METRIC )

    def train( self, dataGenTrain, trainImg, trainLabel, validImg, validLabel ):

        trainSteps = trainImg.shape[0] / BATCH\_SIZE

        validSteps = validImg.shape[0] / BATCH\_SIZE

        self.history = self.model.fit( dataGenTrain.flow( trainImg, trainLabel, batch\_size = BATCH\_SIZE ), epochs = NUM\_EPOCHS,

                    steps\_per\_epoch = trainSteps, validation\_steps = validSteps,

                    validation\_data = ( validImg, validLabel ), callbacks = [ self.lrdecay, cp\_callback ] )

    def test( self, testImg, testLabel ):

        self.model.evaluate( testImg, testLabel )

        return self.model.predict( testImg )

    def load( self, pathName ):

        self.model.load\_weights( pathName )

# Data Augmentation inspired by:

# https://www.tensorflow.org/tutorials/images/data\_augmentation?hl=sv

def resize\_and\_rescale( image ):

    image = tf.cast( image, tf.float32 )

    image = tf.image.resize( image, [IMG\_SIZE, IMG\_SIZE] )

    image = ( image / 255.0 )

    return image

def soften( label, mask ):

    # Get sum of 1s for each bounding box == area of each bounding box

    # Divide by area of image

    alpha = tf.reduce\_sum( mask, [1,2] ) / ( IMG\_SIZE\*\*2 )

    assert all( x <= 1 for x in alpha )

    return ( label \* ( 1-alpha ) + ( 1-label )\*alpha / ( NUM\_CLASSES-1 ) ) \* BETA + ( ( 1-BETA ) \* label )

def augment( image, label, mask ):

    numEx = tf.shape( image )[0]

    image = resize\_and\_rescale( image )

    # Add 6 pixels of padding

    image = tf.image.resize\_with\_crop\_or\_pad( image, IMG\_SIZE + 6, IMG\_SIZE + 6 )

    mask = tf.image.resize\_with\_crop\_or\_pad( mask, IMG\_SIZE + 6, IMG\_SIZE + 6 )

    # Random crop back to the original size

    seedRand = np.random.randint( 10000 )

    image = tf.image.random\_crop( image, size=[ numEx, IMG\_SIZE, IMG\_SIZE, 3 ], seed=seedRand )

    mask = tf.image.random\_crop( mask, size=[ numEx, IMG\_SIZE, IMG\_SIZE, 1 ], seed=seedRand )

    # Adaptive Label Smoothing

    if ALS:

        label = soften( label, mask )

    return image, label, mask

# Underconfidence and Overconfidence based on:

# http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.720.9822

def uncertainty( label ):

    # Ensure not taking log of 0

    assert( np.min( labelProbs > 0 ) )

    temp = label \* tf.math.log( label ) / tf.math.log( tf.convert\_to\_tensor( NUM\_CLASSES, dtype=tf.float32 ) ) # Convert to log base NUM\_CLASSES

    return -tf.reduce\_sum( temp, axis=-1 )  # Sum for each example

def calcConf( labelPreds, labelTruthSparse, ind, labelProbs, labelTruth ):

    numEx = tf.shape( labelProbs )[0]

    K\_c = tf.cast( tf.reduce\_sum( ind ), tf.int32 )         # Gets number of correct predictions

    K\_f = numEx - K\_c

    h = uncertainty( labelProbs )

    confCorrect = ind \* h

    confIncorrect = ( 1-ind ) \* ( 1-h )

    underConf = tf.cast( 1 / K\_c, tf.float32 ) \* tf.reduce\_sum( confCorrect )

    overConf = tf.cast( 1 / K\_f, tf.float32 ) \* tf.reduce\_sum( confIncorrect )

    return underConf, overConf

# ECE and MCE based on:

# https://arxiv.org/pdf/1706.04599.pdf and

# http://people.cs.pitt.edu/~milos/research/AAAI\_Calibration.pdf

def calcCE( labelPreds, labelTruthSparse, ind, labelProbs, labelTruth, binSize ):

    numEx = tf.shape( labelProbs )[0]

    # Split into bins

    unevenSize = numEx.numpy() % binSize    # Get the leftover bin

    labelProbs, unevenProbs = tf.split( labelProbs, [ numEx-unevenSize, unevenSize ] )

    ind, unevenInd = tf.split( ind, [ numEx-unevenSize, unevenSize ] )

    splitProbs = tf.split( labelProbs, binSize )

    splitAcc = tf.split( ind, binSize )

    # For easier computation (list -> tensor)

    splitProbs = tf.transpose( splitProbs, [1,0,2] )

    splitAcc = tf.transpose( splitAcc )

    # Uneven calculations

    unevenAcc = tf.reduce\_sum( unevenInd, axis=-1 ) / unevenSize                        # Gets number of correct predictions

    unevenConf = tf.reduce\_sum( tf.math.reduce\_max( unevenProbs, axis=-1 ), axis=-1 )   # Gets max probability

    unevenConf = unevenConf / unevenSize

    # Bin calculations

    binAcc = tf.reduce\_sum( splitAcc, axis=-1 ) / binSize                           # Gets number of correct predictions

    binConf = tf.reduce\_sum( tf.math.reduce\_max( splitProbs, axis=-1 ), axis=-1 )   # Gets max probability

    binConf = binConf / binSize

    unevenDiff = abs( unevenAcc - unevenConf )

    binDiff = abs( binAcc - binConf )

    ECE = ( tf.reduce\_sum( unevenDiff\*unevenSize ) + tf.reduce\_sum( binDiff\*binSize, axis=-1 ) ) / tf.cast( numEx, tf.float32 )

    MCE = max( tf.reduce\_max( unevenDiff ), tf.reduce\_max( binDiff ) )

    return ECE, MCE

def calcStats( labelProbs, labelTruth ):

    labelPreds = tf.math.argmax( labelProbs, axis=-1 )          # Gets sparse prediction for each example

    labelTruthSparse = tf.math.argmax( labelTruth, axis=-1 )    # Gets sparse label for each example

    ind = ( labelPreds.numpy() == labelTruthSparse.numpy() )    # Get number of correct labels

    ind = tf.cast( ind, tf.float32 )

    underConf, overConf = calcConf( labelPreds, labelTruthSparse, ind, labelProbs, labelTruth )

    ECE15, MCE = calcCE( labelPreds, labelTruthSparse, ind, labelProbs, labelTruth, 15 )

    ECE100, \_ = calcCE( labelPreds, labelTruthSparse, ind, labelProbs, labelTruth, 100 )

    # Display results

    print( "O.Conf: ", overConf )

    print( "U.Conf: ", underConf )

    print( "ECE15: ", ECE15 )

    print( "ECE100: ", ECE100 )

    print( "MCE: ", MCE )

    return ECE15, ECE100, MCE

if \_\_name\_\_ == "\_\_main\_\_":

    # Load data

    (dsTrain, dsValid), info = tfds.load( 'oxford\_iiit\_pet', split=[ 'train', 'test' ], with\_info=True )

    # Test images

    #testLoad( dsTrain, info )

    #testLoad( dsValid, info )

    # Load data into a dictionary

    trainDict = { 'image': [], 'label': [], 'mask': [] }

    validDict = { 'image': [], 'label': [] }

    for example in dsTrain:

        image\_name = example["file\_name"]

        image = example["image"]

        label = example["label"]

        # Get name of file

        name = tf.get\_static\_value( image\_name ).decode( 'utf-8' )

        name = os.path.splitext( name )[0]

        xml\_name = name + ".xml"

        if( xml\_name in missingXMLs ):

            continue

        # Add bounding boxes

        xml\_path = os.path.join( xmls\_dir\_path, xml\_name )

        sizeInfo = []

        for \_, elem in ElementTree.iterparse( xml\_path ):

            unmarshal = unmarshallers.get(elem.tag)

            if unmarshal:

                data = unmarshal(elem)

                elem.clear()

                elem.text = data

                sizeInfo.append( elem.text )

        assert ( len(sizeInfo) == 6 )   # Ensure there aren't multiple bounding boxes

        W, H, w1, h1, w2, h2 = sizeInfo

        # Create mask

        mask = np.zeros( ( H,W ) )

        mask[ h1:h2, w1:w2 ] = 1

        mask = np.expand\_dims( mask, -1 )

        # Resize images

        image = tf.image.resize(image, [IMG\_SIZE, IMG\_SIZE])

        mask = tf.image.resize(mask, [IMG\_SIZE, IMG\_SIZE])

        trainDict[ 'image' ].append( image )

        trainDict[ 'label' ].append( label )

        trainDict[ 'mask' ].append( mask )

    for example in dsValid:

        image = example[ 'image' ]

        label = example[ 'label' ]

        image = resize\_and\_rescale( image )

        validDict[ 'image' ].append( image )

        validDict[ 'label' ].append( label )

    # Clear memory

    dsTrain = None

    dsValid = None

    info = None

    gc.collect()

    print( "Number of training examples:", len( trainDict[ 'image' ] ) )

    assert all( x.shape == (IMG\_SIZE, IMG\_SIZE, 3) for x in trainDict[ 'image' ] )

    # Combine list of arrays into a single array

    trainImg = tf.stack( trainDict[ 'image' ], axis=0 )

    trainLabel = np.hstack( trainDict[ 'label' ] )

    validImg = tf.stack( validDict[ 'image' ], axis=0 )

    validLabel = np.hstack( validDict[ 'label' ] )

    mask = trainDict[ 'mask' ]

    # Clear memory

    trainDict = None

    validDict = None

    gc.collect()

    # One-Hot Encode for ALS

    trainLabel = tf.one\_hot( trainLabel, NUM\_CLASSES )

    validLabel = tf.one\_hot( validLabel, NUM\_CLASSES )

    # Apply random cropping to training data

    trainImg, trainLabel, mask = augment( trainImg, trainLabel, mask )

    if MASKING:

        # Compute pixel-mean mask

        avgVal = tf.reduce\_mean( trainImg, [ 1, 2 ] )

        avgVal = tf.reshape( avgVal, [ -1, 1, 1, 3 ] )

        maskAdd = (1-mask) \* avgVal.numpy()

        # Apply masking

        trainImg = trainImg \* mask

        trainImg = trainImg + maskAdd

    mask = None

    maskAdd = None

    gc.collect()

    # Test images

    for i in range(25):

        plt.subplot( 5, 5, i+1 )

        plt.imshow( trainImg[i] )

    plt.show()

    # Convert to tensor

    trainImg = tf.convert\_to\_tensor( trainImg, dtype=tf.float32 )

    trainLabel = tf.convert\_to\_tensor( trainLabel )

    validImg = tf.convert\_to\_tensor( validImg, dtype=tf.float32 )

    validLabel = tf.convert\_to\_tensor( validLabel )

    # Image Data Generator

    dataGenTrain = tf.keras.preprocessing.image.ImageDataGenerator( rotation\_range=15, width\_shift\_range=0.2, height\_shift\_range=0.2, horizontal\_flip=True )

    dataGenTrain.fit( trainImg )

    # Ensure all values are normalized

    assert all( np.max(x) <= 1 for x in trainImg )

    assert all( np.max(x) <= 1 for x in validImg )

    # Model

    model = imgClassMod()

    if train:

        model.train( dataGenTrain, trainImg, trainLabel, validImg, validLabel )

    if test:

        pathName = os.getcwd() + "\\Checkpoints\\"

        pathName += "Baseline"

        pathName += "\\cp-0200.ckpt"

        print(pathName)

        model.load( pathName )

        labelProbs = tf.convert\_to\_tensor( model.test( validImg, validLabel ) )

        # Calculate metrics

        calcStats( labelProbs, validLabel )

I used the Oxford IIIT Pets dataset. This dataset consists of around 200 images per class for 37 classes. However, the training and test sets were created with a 50-50 split. I initially wanted to recombine them and create my own 80-20 or 90-10 split, but I found that the images in the test set did not have bounding boxes, so I could not use them for training. This meant that for training, I have around 100 images per class for 37 classes, which is a very small amount. I suspected this would easily overfit, so I attempted to mitigate this by using TensorFlow’s ImageDataGenerator to augment the training examples, creating new examples.

I decided to exclude some examples for two main reasons. The first reason is that some of the images were unusable; They were either saved under the incorrect format (.jpg instead of .png or .gif), or they were corrupted. The second reason is that some of the training images did not have proper bounding boxes. Some of them did not have an .xml file, meaning there was no associated bounding box for the image. One of the images had two bounding boxes, and the authors mentioned a strategy (masking) for dealing with images with multiple bounding boxes. However, as there was only one image with more than one bounding box, I decided it would not significantly change the results if it was not included. I excluded this image for consistency and convenience.

I decided not to use two validation sets like the authors did. This is because the V2 validation set that the authors used was a set of “challenging” images compared to the V1 validation set. However, the Oxford IIT Pets dataset does not have a comparable split. Randomly splitting the validation set into two independent sets would have been pointless, as they would both have similar results.

I initially misunderstood what the authors meant by ‘masking’. I initially understood it to mean replacing every pixel outside of the bounding box with a constant value. However, after getting terrible results, I realized that the authors use it to deal with images with multiple bounding boxes. The authors mask every bounding box except for one, which also results in additional training examples. This is not very applicable to the dataset I used, because all of the examples except for one have a single bounding box. I replicated the masking results achieved by the authors without using masking.

I resize every image to 100 by 100 pixels. I initially resized to 150 by 150 pixels, but I decided to further reduce the size to facilitate faster training. I also add 6 pixels to each image and randomly crop it back to 100 by 100 pixels.

I use ResNet50. I train for 200 epochs using SGD with a learning rate of 0.001 decayed by 0.1 at epochs 50, 100, and 150 and a momentum of 0.9. I use a batch size of 16 and L2 regularization with a penalty term of 0.01.

At the end of training, my model achieves over 99% training accuracy, while my validation accuracy is around 47%. Not only is there significant overfitting, but the almost perfect training accuracy suggests that the model is memorizing the training data. I did not take additional measures to combat overfitting as that was not the primary goal of the paper and because the limited examples available in the dataset were likely to result in overfitting.

Table 1: Classification and calibration results with Oxford IIIT Pets

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Method | Train N | ACC | ECE  100 | ECE  15 | MCE | O.conf | U.conf |
|  |  |  |  |  |  |  |  |
| Hard Label | 3.668k | 0.465 | 0.289 | 0.289 | 0.631 | 0.728 | 0.127 |
| A. L. S. | 3.668k | 0.438 | 0.036 | 0.097 | 0.368 | 0.285 | 0.568 |
| A. L. S. (Beta=0.75) | 3.668k | 0.457 | 0.031 | 0.102 | 0.460 | 0.331 | 0.515 |
| A. L. S. (Beta=0.25) | 3.668k | 0.460 | 0.132 | 0.147 | 0.460 | 0.510 | 0.321 |
|  |  |  |  |  |  |  |  |
| Hard Label (incorrect mask) | 3.668k | 0.148 | N/A | N/A | N/A | N/A | N/A |
| A. L. S. (incorrect mask) | 3.668k | 0.227 | N/A | N/A | N/A | N/A | N/A |

The results in Table 1 are mostly consistent with the results achieved in the paper. The Hard Label method is equivalent to beta=0. The A.L.S. method is equivalent to beta=1. With a decreasing beta, the overconfidence increases while the underconfidence decreases. The accuracies achieved are also consistent. The accuracies are mostly consistent, with a slight increase with a decreasing beta.

MCE was calculated along with ECE15. The authors did not mention the bin size used to calculate MCE. Calculating MCE along with ECE100 resulted in a smaller value. It is possible that there is an error in the implementation of the ECE and MCE calculations. The results obtained are the opposite of the results from the paper. In the results, ECE100 is consistently smaller than ECE15, while in the paper, the opposite is true. In the results, ECE and MCE increase with a decreasing beta, while in the paper, the opposite is true.

The results obtained by masking are reported. The Hard Label (incorrect mask) method utilized zero masking, meaning the pixels outside of the bounding box were masked with zeroes. The A.L.S. (incorrect mask) method utilized mean pixel masking, meaning the pixels outside of the bounding box were masked with the average pixel value for each respective channel. The additional statistics were not computed for these incorrect methods.

A.L.S. with beta=0.75 appears to be the best compromise out of all of the methods. It has a small reduction in accuracy, while significantly reducing the overconfidence of the model. This is in contrast with A.L.S. with beta=1, which has a larger decrease in accuracy while barely reducing the overconfidence compared to A.L.S. with beta=0.75. A.L.S. with beta=0.25 achieves almost the same accuracy as A.L.S. with beta=0.75, while also having a significantly higher overconfidence.

Overall, a lot of my problems were caused by the dataset I selected. However, I chose the dataset, so I had to work through the problems. It is very likely that I could have achieved better results with a better dataset. If I had more computing power, I could have resized the images to a large size. Currently, resizing to 100 by 100 pixels discards a lot of information for the larger images.