# construct the argument parse and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-d", "--dataset", required=True,

help="path to input dataset")

ap.add\_argument("-m", "--model", required=True,

help="path to output model")

ap.add\_argument("-p", "--plot", type=str, default="plot.png",

help="path to output loss/accuracy plot")

args = vars(ap.parse\_args())

# initialize the number of epochs to train for, initia learning rate,

# and batch size

#EPOCHS = 1

INIT\_LR = 0.005

#BS = 500

# initialize the data and labels

print("[INFO] loading images...")

data = []

labels = []

# grab the image paths and randomly shuffle them

imagePaths = sorted(list(paths.list\_images(args["dataset"])))

random.seed(42)

random.shuffle(imagePaths)

# loop over the input images

for imagePath in imagePaths:

# load the image, pre-process it, and store it in the data list

image = cv2.imread(imagePath)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

image = cv2.resize(image, (40, 40))

image = img\_to\_array(image)

data.append(image)

# extract the class label from the image path and update the

# labels list

label = imagePath.split(os.path.sep)[-2]

label = 1 if label == "whales" else 0

labels.append(label)

data = np.array(data, dtype="float")

labels = np.array(labels)

print(data.shape)

# partition the data into training and testing splits using 75% of

# the data for training and the remaining 25% for testing

(trainX, testX, trainY, testY) = train\_test\_split(data,

labels, test\_size=0.25, random\_state=42)

print(trainX.shape)

# convert the labels from integers to vectors

trainY = to\_categorical(trainY, num\_classes=2)

testY = to\_categorical(testY, num\_classes=2)

# initialize the model

# Sub-Spectrogram Size

splitSize = 10

# Mel-bins overlap

overlap = 5

# Time Indices

timeInd = 40

melSize=40

# Channels used

channels = 1

height=40

width=40

depth=1

inputShape = (height, width, depth)

outputs = []

# if we are using "channels first", update the input shape

if K.image\_data\_format() == "channels\_first":

inputShape = (depth, height, width)

inputLayer = Input(inputShape)

print(inputLayer.shape)

subSize = int(splitSize / 10)

i = 0

inputs = []

toconcat = []

y\_test = []

y\_train = []

y\_test.append(testY)

y\_train.append(trainY)

while (overlap \* i <= melSize - splitSize):

INPUT = Lambda(lambda inputLayer: inputLayer[:, i \* overlap:i \* overlap + splitSize, :, :],

output\_shape=(splitSize, timeInd, channels))(inputLayer)

#Reshape tensor for 1D modeling

INPUT= Lambda(lambda INPUT:INPUT[:,:,:,-1])(INPUT)

CONV = Conv1D(128, kernel\_size=200, padding='same', kernel\_initializer="he\_normal")(INPUT)

CONV = Activation('relu')(CONV)

outputs.append(CONV)

i=i+1

#print(INPUT.shape)

x = Concatenate()(outputs)

print(x.shape)

x = GRU(128,return\_sequences=True)(x)

print(x.shape)

x = GRU(128,return\_sequences=True)(x)

x = GRU(128,return\_sequences=True)(x)

# return the constructed network architecture

#print(i)

FLATTEN = Flatten()(x)

out = Dense(500, activation='relu')(FLATTEN)

out = Dense(2, activation='sigmoid')(out)

out = Dense(2, activation='softmax')(out)

outputs.append(out)

# construct the image generator for data augmentation

# initialize the model

print("[INFO] compiling model...")

model = Model(inputs=inputLayer,outputs=out)

opt = SGD(lr=INIT\_LR, decay=INIT\_LR / EPOCHS)

model.compile(loss="binary\_crossentropy", optimizer=opt,

metrics=["accuracy"])

# train the network

print("[INFO] training network...")

#H = model.fit(trainX, trainY, batch\_size=BS,

# validation\_data=(testX, testY), steps\_per\_epoch=len(trainX) // BS,

# epochs=EPOCHS, verbose=1)

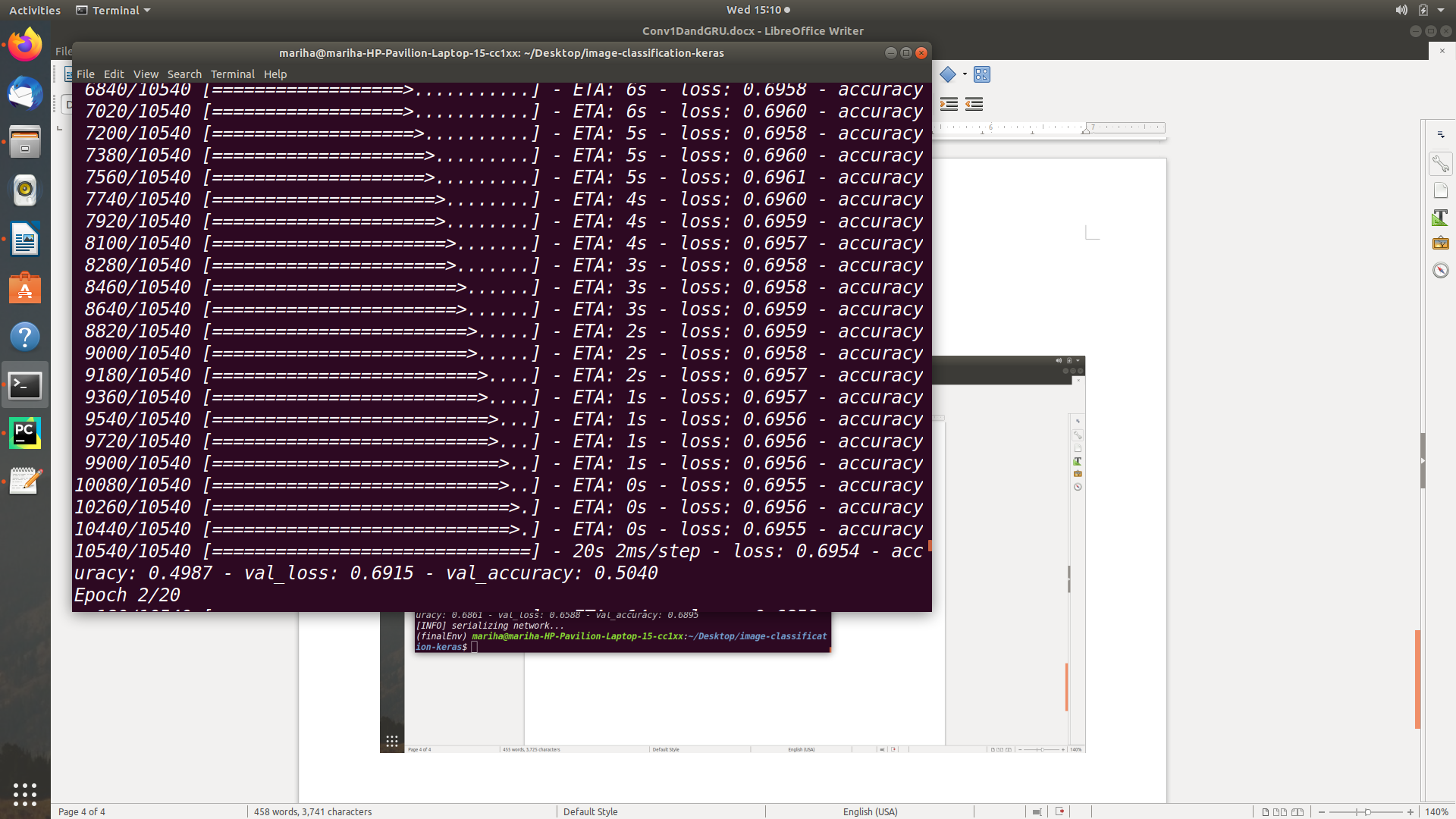
model.fit(trainX, y\_train, batch\_size=180, epochs=20,

verbose=1,validation\_data=(testX, y\_test), shuffle=True)

print("[INFO] serializing network...")

model.save(args["model"])

After 1st epoch:



After 20 epochs

