import argparse, os

import numpy as np

import locale

import tensorflow as tf

import keras

from keras import backend as K

from keras.optimizers import Adam

from keras.models import Model

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization, Conv2D, MaxPooling2D, Input

from keras.optimizers import SGD

from keras.utils import multi\_gpu\_model

from keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

from pathlib import Path

from datetime import datetime as dt

from keras import metrics

from keras.callbacks import EarlyStopping

def create\_cnn(width, height, depth, filters=(16,32,64), regress=False):

# initialize the input shape and channel dimension, assuming

# TensorFlow/channels-last ordering

inputShape = (height, width, depth)

chanDim = -1

# define the model input

inputs = Input(shape=inputShape)

# loop over the number of filters

for (i, f) in enumerate(filters):

# if this is the first CONV layer then set the input

# appropriately

if i == 0:

x = inputs

# CONV => RELU => BN => POOL

x = Conv2D(f, (3, 3), padding="same")(x)

x = Activation("relu")(x)

x = BatchNormalization(axis=chanDim)(x)

x = MaxPooling2D(pool\_size=(2, 2))(x)

x = Dropout(0.5)(x)

# flatten the volume, then FC => RELU => BN => DROPOUT

x = Flatten()(x)

x = Dense(16)(x)

x = Activation("relu")(x)

x = BatchNormalization(axis=chanDim)(x)

x = Dropout(0.5)(x)

# apply another FC layer, this one to match the number of nodes

# coming out of the MLP

x = Dense(4)(x)

x = Activation("relu")(x)

# check to see if the regression node should be added

if regress:

x = Dense(1, activation="linear")(x)

# construct the CNN

model = Model(inputs, x)

return model

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument('--epochs', type=int, default=10)

parser.add\_argument('--learning-rate', type=float, default=0.01)

parser.add\_argument('--batch-size', type=int, default=128)

parser.add\_argument('--gpu-count', type=int, default=os.environ['SM\_NUM\_GPUS'])

parser.add\_argument('--model-dir', type=str, default=os.environ['SM\_MODEL\_DIR'])

parser.add\_argument('--training', type=str, default=os.environ['SM\_CHANNEL\_TRAINING'])

parser.add\_argument('--validation', type=str, default=os.environ['SM\_CHANNEL\_VALIDATION'])

parser.add\_argument('--output-dir', default=os.getenv('SM\_OUTPUT\_DATA\_DIR', 'outputs/'))

args, \_ = parser.parse\_known\_args()

epochs = args.epochs

lr = args.learning\_rate

batch\_size = args.batch\_size

gpu\_count = args.gpu\_count

model\_dir = args.model\_dir

training\_dir = args.training

validation\_dir = args.validation

is\_sagemaker = 'SM\_CHANNEL\_DATASET' in os.environ

# output directory

output\_dir = Path(args.output\_dir)

if is\_sagemaker:

model\_dir = args.model\_dir

else:

output\_dir /= dt.now().strftime('%Y-%m-%d-%H-%M')

output\_dir.mkdir(parents=True)

model\_dir = str(output\_dir / args.model\_dir)

x\_train = np.load(os.path.join(training\_dir, 'training.npz'))['image']

y\_train = np.load(os.path.join(training\_dir, 'training.npz'))['label']

x\_val = np.load(os.path.join(validation\_dir, 'validation.npz'))['image']

y\_val = np.load(os.path.join(validation\_dir, 'validation.npz'))['label']

# input image dimensions

img\_rows, img\_cols = 64, 64

# Tensorflow needs image channels last, e.g. (batch size, width, height, channels)

K.set\_image\_data\_format('channels\_last')

print(K.image\_data\_format())

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

print("Incorrect configuration: Tensorflow needs channels\_last")

else:

# channels last

#x\_train = x\_train.reshape(x\_train.shape[0], img\_rows, img\_cols, 1)

#x\_val = x\_val.reshape(x\_val.shape[0], img\_rows, img\_cols, 1)

input\_shape = (img\_rows, img\_cols, 3)

batch\_norm\_axis=-1

print('x\_train shape:', x\_train.shape)

print(x\_train.shape[0], 'train samples')

print(x\_val.shape[0], 'test samples')

# Normalize pixel values

x\_train = x\_train.astype('float32')

x\_val = x\_val.astype('float32')

y\_train = y\_train.astype('float32')

y\_val = y\_val.astype('float32')

x\_train /= 255

x\_val /= 255

# Convert class vectors to binary class matrices

#num\_classes = 10

#y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

#y\_val = keras.utils.to\_categorical(y\_val, num\_classes)

model = create\_cnn(64, 64, 3, regress=True)

print(model.summary())

aug = ImageDataGenerator(horizontal\_flip=True,

zoom\_range=[0.9,1.1],

fill\_mode="nearest",

brightness\_range=[0.5,1.25],

width\_shift\_range=[-8,8],

height\_shift\_range=[-8,8])

if gpu\_count > 1:

model = multi\_gpu\_model(model, gpus=gpu\_count)

model.compile(loss="mean\_absolute\_error",

optimizer=Adam(lr=1e-3, decay=1e-4))

# simple early stopping

es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, patience=50)

history = model.fit\_generator(aug.flow(x\_train, y\_train, batch\_size=batch\_size),

validation\_data=aug.flow(x\_val, y\_val, batch\_size=batch\_size),

validation\_steps=len(x\_val)/batch\_size,

steps\_per\_epoch=len(x\_train)/batch\_size,

epochs=epochs,

verbose=2,

callbacks=[es])

# make predictions on the testing data

print("[INFO] predicting pH levels...")

preds = model.predict(x\_val)

# compute the difference between the \*predicted\* house prices and the

# \*actual\* house prices, then compute the percentage difference and

# the absolute percentage difference

diff = preds.flatten() - y\_val

percentDiff = (diff / y\_val) \* 100

absPercentDiff = np.abs(percentDiff)

# compute the mean and standard deviation of the absolute percentage

# difference

mean = np.mean(absPercentDiff)

std = np.std(absPercentDiff)

# finally, show some statistics on our model

#locale.setlocale(locale.LC\_ALL, "en\_US.UTF-8")

#print("[INFO] avg. house price: {}, std house price: {}".format(

#locale.currency(df["price"].mean(), grouping=True),

#locale.currency(df["price"].std(), grouping=True)))

print("[INFO] mean: {:.2f}%, std: {:.2f}%".format(mean, std))

print(preds)

print(y\_val)

# save Keras model for Tensorflow Serving

sess = K.get\_session()

tf.saved\_model.simple\_save(

sess,

os.path.join(model\_dir, 'model/1'),

inputs={'inputs': model.input},

outputs={t.name: t for t in model.outputs})

print(history.history.keys())

# summarize history for loss

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('model loss')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper left')

loss\_file = output\_dir / 'loss.png'

plt.savefig(str(loss\_file))

plt.clf()