"""

After extracting the RAR, we run this to move all the files into

the appropriate train/test folders.

Should only run this file once!

"""

import os

import os.path

def get\_train\_file(version='01'):

train\_file = os.path.join('data/TrainTestlist', 'trainlist' + version + '.txt')

return train\_file

def get\_test\_file(version='01'):

test\_file = os.path.join('data/TrainTestlist', 'testlist' + version + '.txt')

return test\_file

def generate\_train\_test\_files(version='01'):

"""

Dynamic generate train/test file as a name list txt file.

"""

path\_annotation = 'trainingset\_annotations.txt'

path\_train = './data/rtvcdata/train'

path\_test = './data/rtvcdata/test'

## clear file.

open(get\_train\_file(), 'w').close()

open(get\_test\_file(), 'w').close()

###

files\_train = os.listdir(path\_train)

print("files\_train: %s", files\_train)

for name in files\_train:

with open(path\_annotation,'r') as fin:

lines = fin.readlines()

txt\_file\_train = open(get\_train\_file(), "a")

for line in lines:

if line.find(name)!=-1:

print("found match line: %s, name: %s, then write", line, name)

### write it to train list text file line by line

txt\_file\_train.write("%s" % line)

txt\_file\_train.close()

###

files\_test = os.listdir(path\_test)

print("files\_test: %s", files\_test)

for name in files\_test:

with open(path\_annotation,'r') as fin:

lines = fin.readlines()

txt\_file\_test = open(get\_test\_file(), "a")

for line in lines:

if line.find(name)!=-1:

print("found match line: %s, name: %s, then write", line, name)

### write it to train list text file line by line

txt\_file\_test.write("%s" % line)

txt\_file\_test.close()

def main():

"""

Go through each of our train/test text files and move the videos

to the right place.

"""

###!! Generate dataset at first.

generate\_train\_test\_files()

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

main()

"""

After extracting the RAR, we run this to move all the files into

the appropriate train/test folders.

Should only run this file once!

"""

import os

import os.path

#@see: https://www.techbeamers.com/python-copy-file/#shutil-copyfile

from shutil import copyfile

from sys import exit

def get\_train\_file(version='01'):

train\_file = os.path.join('data/TrainTestlist', 'trainlist' + version + '.txt')

return train\_file

def get\_test\_file(version='01'):

test\_file = os.path.join('data/TrainTestlist', 'testlist' + version + '.txt')

return test\_file

def get\_train\_test\_lists(version='01'):

"""

Using one of the train/test files (01, 02, or 03), get the filename

breakdowns we'll later use to move everything.

"""

# Get our files based on version.

test\_file = get\_test\_file(version)

train\_file = get\_train\_file(version)

# Build the test list.

with open(test\_file) as fin:

test\_list = [row.strip() for row in list(fin)]

test\_list = [row.split(' ')[0] for row in test\_list]

# Build the train list. Extra step to remove the class index.

with open(train\_file) as fin:

train\_list = [row.strip() for row in list(fin)]

train\_list = [row.split(' ')[0] for row in train\_list]

# Set the groups in a dictionary.

file\_groups = {

'train': train\_list,

'test': test\_list

}

return file\_groups

def move\_files(file\_groups):

"""This assumes all of our files are currently in \_this\_ directory.

So move them to the appropriate spot. Only needs to happen once.

"""

# Do each of our groups.

for group, videos in file\_groups.items():

# Do each of our videos.

for video in videos:

print(video)

# Get the parts.

#parts = video.split(os.path.sep)

parts = video.split(",")

#print(parts)

classname = parts[1]

#classname = ""

#print(classname)

filename = parts[0]

print(classname,filename)

# Check if this class exists.

# if not os.path.exists(os.path.join(group, classname)):

# print("Creating folder for %s/%s" % (group, classname))

# os.makedirs(os.path.join(group, classname))

# Check if we have already moved this file, or at least that it

# exists to move.

fullfilename = "data/rtvcdata/" + group + "/"+ filename

print(fullfilename)

if not os.path.exists(fullfilename):

print("Can't find %s to move. Skipping." % (fullfilename))

continue

# Move it.

dest = os.path.join('data/',group, filename)

print("Copying %s to %s" % (fullfilename, dest))

#os.rename(fullfilename, dest)

copyfile(fullfilename, dest)

print("Done.")

def main():

"""

Go through each of our train/test text files and move the videos

to the right place.

"""

# Get the videos in groups so we can move them.

group\_lists = get\_train\_test\_lists()

#print(group\_lists)

# Move the files.

move\_files(group\_lists)

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

main()

"""

After moving all the files using the 1\_ file, we run this one to extract

the images from the videos and also create a data file we can use

for training and testing later.

"""

import csv

import glob

import os

import os.path

from subprocess import call

def extract\_files():

"""After we have all of our videos split between train and test, and

all nested within folders representing their classes, we need to

make a data file that we can reference when training our RNN(s).

This will let us keep track of image sequences and other parts

of the training process.

We'll first need to extract images from each of the videos. We'll

need to record the following data in the file:

[train|test], class, filename, nb frames

Extracting can be done with ffmpeg:

`ffmpeg -i video.mpg image-%04d.jpg`

"""

data\_file = []

folders = ['train', 'test']

for folder in folders:

class\_folders = glob.glob(os.path.join('data', folder, '\*'))

#print(class\_folders)

for vid\_class in class\_folders:

#print(vid\_class)

#class\_files = glob.glob(os.path.join(vid\_class, '\*.mp4'))

#class\_files = vid\_class

#for video\_path in class\_files:

# Get the parts of the file.

video\_parts = get\_video\_parts(vid\_class)

train\_or\_test, classname, filename\_no\_ext, filename = video\_parts

# Only extract if we haven't done it yet. Otherwise, just get

# the info.

if not check\_already\_extracted(video\_parts):

# Now extract it.

src = os.path.join('data', train\_or\_test, filename)

dest = os.path.join('data', train\_or\_test,

filename\_no\_ext + '-%04d.jpg')

#print(src,dest)

call(["ffmpeg", "-i", src, dest])

# Now get how many frames it is.

nb\_frames = get\_nb\_frames\_for\_video(video\_parts)

data\_file.append([train\_or\_test, classname, filename\_no\_ext, nb\_frames])

print("Generated %d frames for %s" % (nb\_frames, filename\_no\_ext))

## clear file.

open('data\_file.csv', 'w').close()

with open('data\_file.csv', 'w') as fout:

writer = csv.writer(fout)

writer.writerows(data\_file)

print("Extracted and wrote %d video files." % (len(data\_file)))

def get\_nb\_frames\_for\_video(video\_parts):

"""Given video parts of an (assumed) already extracted video, return

the number of frames that were extracted."""

train\_or\_test, classname, filename\_no\_ext, \_ = video\_parts

generated\_files = glob.glob(os.path.join('data', train\_or\_test,

filename\_no\_ext + '\*.jpg'))

return len(generated\_files)

def get\_video\_classes(video\_name):

"""find video classes from anotation txt file."""

path\_annotation = 'trainingset\_annotations.txt'

with open(path\_annotation,'r') as fin:

lines = fin.readlines()

for line in lines:

if line.find(video\_name)!=-1:

print("found match line: %s, video\_name: %s, then write", line, video\_name)

parts = line.split(",")

classnames = "" # e.g: c1\_c2\_...

for i in range(1, len(parts)):#without filename as index 0

classnames += "\_" + parts[i].strip("\n")

print(classnames)

return classnames

def get\_video\_parts(video\_path):

"""Given a full path to a video, return its parts."""

parts = video\_path.split(os.path.sep)

#parts = video\_path.split("\\")

print(parts)

filename = parts[2]

#print(filename)

classname = get\_video\_classes(filename)

print(classname)

filename\_no\_ext = filename.split('.')[0] + classname

print(filename\_no\_ext)

train\_or\_test = parts[1]

return train\_or\_test, classname, filename\_no\_ext, filename

def check\_already\_extracted(video\_parts):

"""Check to see if we created the -0001 frame of this file."""

train\_or\_test, classname, filename\_no\_ext, \_ = video\_parts

return bool(os.path.exists(os.path.join(train\_or\_test, classname,

filename\_no\_ext + '-0001.jpg')))

def main():

"""

Extract images from videos and build a new file that we

can use as our data input file. It can have format:

[train|test], class, filename, nb frames

"""

extract\_files()

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

main()

"""

This script generates extracted features for each video, which other

models make use of.

You can change you sequence length and limit to a set number of classes

below.

class\_limit is an integer that denotes the first N classes you want to

extract features from. This is useful is you don't want to wait to

extract all 101 classes. For instance, set class\_limit = 8 to just

extract features for the first 8 (alphabetical) classes in the dataset.

Then set the same number when training models.

"""

import numpy as np

import os.path

from data import DataSet

from extractor import Extractor

from tqdm import tqdm

# Set defaults.

seq\_length = 40

class\_limit = 63 # Number of classes to extract. Can be 1-101 or None for all.

# Get the dataset.

data = DataSet(seq\_length=seq\_length, class\_limit=class\_limit)

# get the model.

model = Extractor()

# Loop through data.

print(data.data)

pbar = tqdm(total=len(data.data))

for video in data.data:

print(data.data,video)

# Get the path to the sequence for this video.

path = os.path.join('data', 'sequences', str(video[2]) + '-' + str(seq\_length) + \

'-features') # numpy will auto-append .npy

print(path)

# Check if we already have it.

if os.path.isfile(path + '.npy'):

pbar.update(1)

continue

# Get the frames for this video.

#print(video)

frames = data.get\_frames\_for\_sample(video)

print("frames: %s",frames)

# Now downsample to just the ones we need.

frames = data.rescale\_list(frames, seq\_length)

print("downsampled frames: %s",frames)

# Now loop through and extract features to build the sequence.

sequence = []

for image in frames:

features = model.extract(image)

sequence.append(features)

# Save the sequence.

np.save(path, sequence)

pbar.update(1)

pbar.close()

from keras.preprocessing import image

from keras.applications.inception\_v3 import InceptionV3, preprocess\_input

from keras.models import Model, load\_model

from keras.layers import Input

import numpy as np

class Extractor():

def \_\_init\_\_(self, weights=None):

"""Either load pretrained from imagenet, or load our saved

weights from our own training."""

self.weights = weights # so we can check elsewhere which model

if weights is None:

# Get model with pretrained weights.

base\_model = InceptionV3(

weights='imagenet',

include\_top=True

)

# We'll extract features at the final pool layer.

self.model = Model(

inputs=base\_model.input,

outputs=base\_model.get\_layer('avg\_pool').output

)

else:

# Load the model first.

self.model = load\_model(weights)

# Then remove the top so we get features not predictions.

# From: https://github.com/fchollet/keras/issues/2371

self.model.layers.pop()

self.model.layers.pop() # two pops to get to pool layer

self.model.outputs = [self.model.layers[-1].output]

self.model.output\_layers = [self.model.layers[-1]]

self.model.layers[-1].outbound\_nodes = []

def extract(self, image\_path):

img = image.load\_img(image\_path, target\_size=(299, 299))

x = image.img\_to\_array(img)

x = np.expand\_dims(x, axis=0)

x = preprocess\_input(x)

# Get the prediction.

features = self.model.predict(x)

if self.weights is None:

# For imagenet/default network:

features = features[0]

else:

# For loaded network:

features = features[0]

return features

"""

A collection of models we'll use to attempt to classify videos.

"""

from keras.layers import Dense, Flatten, Dropout, ZeroPadding3D

from keras.layers.recurrent import LSTM

from keras.models import Sequential, load\_model

from keras.optimizers import Adam, RMSprop

from keras.layers.wrappers import TimeDistributed

from keras.layers.convolutional import (Conv2D, MaxPooling3D, Conv3D,

MaxPooling2D)

from collections import deque

import sys

class ResearchModels():

def \_\_init\_\_(self, nb\_classes, model, seq\_length,

saved\_model=None, features\_length=2048):

"""

`model` = one of:

lstm

lrcn

mlp

conv\_3d

c3d

`nb\_classes` = the number of classes to predict

`seq\_length` = the length of our video sequences

`saved\_model` = the path to a saved Keras model to load

"""

# Set defaults.

self.seq\_length = seq\_length

self.load\_model = load\_model

self.saved\_model = saved\_model

self.nb\_classes = nb\_classes

self.feature\_queue = deque()

# Set the metrics. Only use top k if there's a need.

metrics = ['accuracy']

if self.nb\_classes >= 10:

metrics.append('top\_k\_categorical\_accuracy')

# Get the appropriate model.

if self.saved\_model is not None:

print("Loading model %s" % self.saved\_model)

self.model = load\_model(self.saved\_model)

elif model == 'lstm':

print("Loading LSTM model.")

self.input\_shape = (seq\_length, features\_length)

self.model = self.lstm()

elif model == 'lrcn':

print("Loading CNN-LSTM model.")

self.input\_shape = (seq\_length, 80, 80, 3)

self.model = self.lrcn()

elif model == 'mlp':

print("Loading simple MLP.")

self.input\_shape = (seq\_length, features\_length)

self.model = self.mlp()

elif model == 'conv\_3d':

print("Loading Conv3D")

self.input\_shape = (seq\_length, 80, 80, 3)

self.model = self.conv\_3d()

elif model == 'c3d':

print("Loading C3D")

self.input\_shape = (seq\_length, 80, 80, 3)

self.model = self.c3d()

else:

print("Unknown network.")

sys.exit()

# Now compile the network.

optimizer = Adam(lr=1e-5, decay=1e-6)

self.model.compile(loss='categorical\_crossentropy', optimizer=optimizer,

metrics=metrics)

print(self.model.summary())

def lstm(self):

"""Build a simple LSTM network. We pass the extracted features from

our CNN to this model predomenently."""

# Model.

model = Sequential()

model.add(LSTM(2048, return\_sequences=False,

input\_shape=self.input\_shape,

dropout=0.5))

model.add(Dense(512, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(self.nb\_classes, activation='softmax'))

return model

def lrcn(self):

"""Build a CNN into RNN.

Starting version from:

https://github.com/udacity/self-driving-car/blob/master/

steering-models/community-models/chauffeur/models.py

Heavily influenced by VGG-16:

https://arxiv.org/abs/1409.1556

Also known as an LRCN:

https://arxiv.org/pdf/1411.4389.pdf

"""

model = Sequential()

model.add(TimeDistributed(Conv2D(32, (7, 7), strides=(2, 2),

activation='relu', padding='same'), input\_shape=self.input\_shape))

model.add(TimeDistributed(Conv2D(32, (3,3),

kernel\_initializer="he\_normal", activation='relu')))

model.add(TimeDistributed(MaxPooling2D((2, 2), strides=(2, 2))))

model.add(TimeDistributed(Conv2D(64, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(Conv2D(64, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(MaxPooling2D((2, 2), strides=(2, 2))))

model.add(TimeDistributed(Conv2D(128, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(Conv2D(128, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(MaxPooling2D((2, 2), strides=(2, 2))))

model.add(TimeDistributed(Conv2D(256, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(Conv2D(256, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(MaxPooling2D((2, 2), strides=(2, 2))))

model.add(TimeDistributed(Conv2D(512, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(Conv2D(512, (3,3),

padding='same', activation='relu')))

model.add(TimeDistributed(MaxPooling2D((2, 2), strides=(2, 2))))

model.add(TimeDistributed(Flatten()))

model.add(Dropout(0.5))

model.add(LSTM(256, return\_sequences=False, dropout=0.5))

model.add(Dense(self.nb\_classes, activation='softmax'))

return model

def mlp(self):

"""Build a simple MLP. It uses extracted features as the input

because of the otherwise too-high dimensionality."""

# Model.

model = Sequential()

model.add(Flatten(input\_shape=self.input\_shape))

model.add(Dense(512))

model.add(Dropout(0.5))

model.add(Dense(512))

model.add(Dropout(0.5))

model.add(Dense(self.nb\_classes, activation='softmax'))

return model

def conv\_3d(self):

"""

Build a 3D convolutional network, based loosely on C3D.

https://arxiv.org/pdf/1412.0767.pdf

"""

# Model.

model = Sequential()

model.add(Conv3D(

32, (3,3,3), activation='relu', input\_shape=self.input\_shape

))

model.add(MaxPooling3D(pool\_size=(1, 2, 2), strides=(1, 2, 2)))

model.add(Conv3D(64, (3,3,3), activation='relu'))

model.add(MaxPooling3D(pool\_size=(1, 2, 2), strides=(1, 2, 2)))

model.add(Conv3D(128, (3,3,3), activation='relu'))

model.add(Conv3D(128, (3,3,3), activation='relu'))

model.add(MaxPooling3D(pool\_size=(1, 2, 2), strides=(1, 2, 2)))

model.add(Conv3D(256, (2,2,2), activation='relu'))

model.add(Conv3D(256, (2,2,2), activation='relu'))

model.add(MaxPooling3D(pool\_size=(1, 2, 2), strides=(1, 2, 2)))

model.add(Flatten())

model.add(Dense(1024))

model.add(Dropout(0.5))

model.add(Dense(1024))

model.add(Dropout(0.5))

model.add(Dense(self.nb\_classes, activation='softmax'))

return model

def c3d(self):

"""

Build a 3D convolutional network, aka C3D.

https://arxiv.org/pdf/1412.0767.pdf

With thanks:

https://gist.github.com/albertomontesg/d8b21a179c1e6cca0480ebdf292c34d2

"""

model = Sequential()

# 1st layer group

model.add(Conv3D(64, 3, 3, 3, activation='relu',

border\_mode='same', name='conv1',

subsample=(1, 1, 1),

input\_shape=self.input\_shape))

model.add(MaxPooling3D(pool\_size=(1, 2, 2), strides=(1, 2, 2),

border\_mode='valid', name='pool1'))

# 2nd layer group

model.add(Conv3D(128, 3, 3, 3, activation='relu',

border\_mode='same', name='conv2',

subsample=(1, 1, 1)))

model.add(MaxPooling3D(pool\_size=(2, 2, 2), strides=(2, 2, 2),

border\_mode='valid', name='pool2'))

# 3rd layer group

model.add(Conv3D(256, 3, 3, 3, activation='relu',

border\_mode='same', name='conv3a',

subsample=(1, 1, 1)))

model.add(Conv3D(256, 3, 3, 3, activation='relu',

border\_mode='same', name='conv3b',

subsample=(1, 1, 1)))

model.add(MaxPooling3D(pool\_size=(2, 2, 2), strides=(2, 2, 2),

border\_mode='valid', name='pool3'))

# 4th layer group

model.add(Conv3D(512, 3, 3, 3, activation='relu',

border\_mode='same', name='conv4a',

subsample=(1, 1, 1)))

model.add(Conv3D(512, 3, 3, 3, activation='relu',

border\_mode='same', name='conv4b',

subsample=(1, 1, 1)))

model.add(MaxPooling3D(pool\_size=(2, 2, 2), strides=(2, 2, 2),

border\_mode='valid', name='pool4'))

# 5th layer group

model.add(Conv3D(512, 3, 3, 3, activation='relu',

border\_mode='same', name='conv5a',

subsample=(1, 1, 1)))

model.add(Conv3D(512, 3, 3, 3, activation='relu',

border\_mode='same', name='conv5b',

subsample=(1, 1, 1)))

model.add(ZeroPadding3D(padding=(0, 1, 1)))

model.add(MaxPooling3D(pool\_size=(2, 2, 2), strides=(2, 2, 2),

border\_mode='valid', name='pool5'))

model.add(Flatten())

# FC layers group

model.add(Dense(4096, activation='relu', name='fc6'))

model.add(Dropout(0.5))

model.add(Dense(4096, activation='relu', name='fc7'))

model.add(Dropout(0.5))

model.add(Dense(self.nb\_classes, activation='softmax'))

return model

"""

Class for managing our data.

"""

import csv

import numpy as np

import random

import glob

import os.path

import sys

import operator

import threading

from processor import process\_image

from keras.utils import to\_categorical

import pandas

import tensorflow as tf

config = tf.ConfigProto()

config.gpu\_options.allow\_growth = True

sess = tf.Session(config=config)

class threadsafe\_iterator:

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

self.iterator = iterator

self.lock = threading.Lock()

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

with self.lock:

return next(self.iterator)

def threadsafe\_generator(func):

"""Decorator"""

def gen(\*a, \*\*kw):

return threadsafe\_iterator(func(\*a, \*\*kw))

return gen

class DataSet():

def \_\_init\_\_(self, seq\_length=40, class\_limit=None, image\_shape=(224, 224, 3)):

"""Constructor.

seq\_length = (int) the number of frames to consider

class\_limit = (int) number of classes to limit the data to.

None = no limit.

"""

self.seq\_length = seq\_length

self.class\_limit = class\_limit

self.sequence\_path = os.path.join('data', 'sequences')

self.max\_frames = 300 # max number of frames a video can have for us to use it

# Get the data.

self.data = self.get\_data()

# Get the classes.

self.classes = self.get\_classes()

print("classes: ", self.classes)

# Now do some minor data cleaning.

self.data = self.clean\_data()

self.image\_shape = image\_shape

@staticmethod

def get\_data():

"""Load our data from file."""

#with open(os.path.join('data', 'data\_file.csv'), 'r') as fin:

# reader = csv.reader(fin)

# data = list(reader)

with open(os.path.join('data\_file.csv'), 'r') as fin:

data = pandas.read\_csv(fin, header=None)

print("get\_data(): ", data.values.tolist())

data = data.values.tolist()

return data

def clean\_data(self):

"""Limit samples to greater than the sequence length and fewer

than N frames. Also limit it to classes we want to use."""

data\_clean = []

for item in self.data:

if int(item[3]) >= self.seq\_length and int(item[3]) <= self.max\_frames \

and item[1].lstrip('\_') in self.classes:

data\_clean.append(item)

return data\_clean

def get\_classes(self):

"""Extract the classes from our data. If we want to limit them,

only return the classes we need."""

classes = []

for item in self.data:

rawitem = item[1].lstrip('\_')

if rawitem not in classes:

classes.append(rawitem)

# Sort them.

classes = sorted(classes)

# Return.

if self.class\_limit is not None:

return classes[:self.class\_limit]

else:

return classes

def get\_class\_one\_hot(self, class\_str):

"""Given a class as a string, return its number in the classes

list. This lets us encode and one-hot it for training."""

# Encode it first.

label\_encoded = self.classes.index(class\_str.lstrip("\_"))

# Now one-hot it.

label\_hot = to\_categorical(label\_encoded, len(self.classes))

assert len(label\_hot) == len(self.classes)

return label\_hot

def split\_train\_test(self):

"""Split the data into train and test groups."""

train = []

test = []

for item in self.data:

print("self.data:, item: ", self.data, item)

if item[0] == 'train':

train.append(item)

else:

test.append(item)

print("split\_train\_test:", train, test)

return train, test

def get\_all\_sequences\_in\_memory(self, train\_test, data\_type):

"""

This is a mirror of our generator, but attempts to load everything into

memory so we can train way faster.

"""

# Get the right dataset.

print("get\_all\_sequences\_in\_memory called.")

train, test = self.split\_train\_test()

data = train if train\_test == 'train' else test

print("Loading %d samples into memory for %sing." % (len(data), train\_test))

X, y = [], []

for row in data:

if data\_type == 'images':

frames = self.get\_frames\_for\_sample(row)

frames = self.rescale\_list(frames, self.seq\_length)

# Build the image sequence

sequence = self.build\_image\_sequence(frames)

else:

sequence = self.get\_extracted\_sequence(data\_type, row)

if sequence is None:

print("Can't find sequence. Did you generate them?")

raise

X.append(sequence)

y.append(self.get\_class\_one\_hot(row[1]))

return np.array(X), np.array(y)

@threadsafe\_generator

def frame\_generator(self, batch\_size, train\_test, data\_type):

"""Return a generator that we can use to train on. There are

a couple different things we can return:

data\_type: 'features', 'images'

"""

# Get the right dataset for the generator.

train, test = self.split\_train\_test()

data = train if train\_test == 'train' else test

print("Creating %s generator with %d samples." % (train\_test, len(data)))

while 1:

X, y = [], []

# Generate batch\_size samples.

for \_ in range(batch\_size):

# Reset to be safe.

sequence = None

# Get a random sample.

sample = random.choice(data)

# Check to see if we've already saved this sequence.

if data\_type is "images":

# Get and resample frames.

frames = self.get\_frames\_for\_sample(sample)

frames = self.rescale\_list(frames, self.seq\_length)

# Build the image sequence

sequence = self.build\_image\_sequence(frames)

else:

# Get the sequence from disk.

sequence = self.get\_extracted\_sequence(data\_type, sample)

if sequence is None:

raise ValueError("Can't find sequence. Did you generate them?")

X.append(sequence)

y.append(self.get\_class\_one\_hot(sample[1]))

yield np.array(X), np.array(y)

def build\_image\_sequence(self, frames):

"""Given a set of frames (filenames), build our sequence."""

return [process\_image(x, self.image\_shape) for x in frames]

def get\_extracted\_sequence(self, data\_type, sample):

"""Get the saved extracted features."""

filename = sample[2]

#print(sample,filename, data\_type)

path = os.path.join(self.sequence\_path, str(filename) + '-' + str(self.seq\_length) + \

'-' + data\_type + '.npy')

print("get\_extracted\_sequence:",path)

if os.path.isfile(path):

return np.load(path)

else:

return None

def get\_frames\_by\_filename(self, filename, data\_type):

print("get\_frames\_by\_filename called.")

"""Given a filename for one of our samples, return the data

the model needs to make predictions."""

# First, find the sample row.

sample = None

print(self.data)

for row in self.data:

if row[2].split("\_")[0] == filename:

sample = row

break

if sample is None:

raise ValueError("Couldn't find sample: %s" % filename)

if data\_type == "images":

# Get and resample frames.

frames = self.get\_frames\_for\_sample(sample)

print("frames: %s,seq\_length: %s" % (frames,self.seq\_length))

frames = self.rescale\_list(frames, self.seq\_length)

# Build the image sequence

sequence = self.build\_image\_sequence(frames)

else:

# Get the sequence from disk.

sequence = self.get\_extracted\_sequence(data\_type, sample)

if sequence is None:

raise ValueError("Can't find sequence. Did you generate them?")

print(sequence)

return sequence

@staticmethod

def get\_frames\_for\_sample(sample):

"""Given a sample row from the data file, get all the corresponding frame

filenames."""

path = os.path.join('data', sample[0])

filename = str(sample[2])

images = sorted(glob.glob(os.path.join(path, filename + '\*jpg')))

return images

@staticmethod

def get\_filename\_from\_image(filename):

parts = filename.split(os.path.sep)

return parts[-1].replace('.jpg', '')

@staticmethod

def rescale\_list(input\_list, size):

"""Given a list and a size, return a rescaled/samples list. For example,

if we want a list of size 5 and we have a list of size 25, return a new

list of size five which is every 5th element of the origina list."""

print("input\_list: %s,size: %s" % (input\_list, size))

assert len(input\_list) >= size

# Get the number to skip between iterations.

skip = len(input\_list) // size

# Build our new output.

output = [input\_list[i] for i in range(0, len(input\_list), skip)]

# Cut off the last one if needed.

return output[:size]

def print\_class\_from\_prediction(self, predictions, nb\_to\_return=5):

"""Given a prediction, print the top classes."""

# Get the prediction for each label.

label\_predictions = {}

for i, label in enumerate(self.classes):

label\_predictions[label] = predictions[i]

# Now sort them.

sorted\_lps = sorted(

label\_predictions.items(),

key=operator.itemgetter(1),

reverse=True

)

# And return the top N.

for i, class\_prediction in enumerate(sorted\_lps):

if i > nb\_to\_return - 1 or class\_prediction[1] == 0.0:

break

print("%s: %.2f" % (class\_prediction[0], class\_prediction[1]))

"""

Given a video path and a saved model (checkpoint), produce classification

predictions.

Note that if using a model that requires features to be extracted, those

features must be extracted first.

Note also that this is a rushed demo script to help a few people who have

requested it and so is quite "rough". :)

"""

from keras.models import load\_model

from data import DataSet

import numpy as np

import time

from datetime import datetime

def predict(data\_type, seq\_length, saved\_model, image\_shape, video\_name, class\_limit):

model = load\_model(saved\_model)

# Get the data and process it.

if image\_shape is None:

data = DataSet(seq\_length=seq\_length, class\_limit=class\_limit)

else:

data = DataSet(seq\_length=seq\_length, image\_shape=image\_shape,

class\_limit=class\_limit)

# Extract the sample from the data.

sample = data.get\_frames\_by\_filename(video\_name, data\_type)

# Predict!

prediction = model.predict(np.expand\_dims(sample, axis=0))

print(prediction)

data.print\_class\_from\_prediction(np.squeeze(prediction, axis=0))

def main():

# model can be one of lstm, lrcn, mlp, conv\_3d, c3d.

#model = 'lstm'

model = 'lstm'

# Must be a weights file.

#saved\_model = 'data/checkpoints/inception.016-1.46.hdf5'

saved\_model = 'data/checkpoints/lstm-features.001-1.733.hdf5'

#saved\_model = 'data/checkpoints/lstm-features.026-0.239.hdf5'

# Sequence length must match the lengh used during training.

seq\_length = 40

# Limit must match that used during training.

class\_limit = 101

# Demo file. Must already be extracted & features generated (if model requires)

# Do not include the extension.

# Assumes it's in data/[train|test]/

# It also must be part of the train/test data.

# TODO Make this way more useful. It should take in the path to

# an actual video file, extract frames, generate sequences, etc.

video\_name = '1000370'

#ideo\_name = 'v\_ApplyLipstick\_g01\_c01'

#video\_name = 'v\_YoYo\_g04\_c02'

# Chose images or features and image shape based on network.

if model in ['conv\_3d', 'c3d', 'lrcn']:

data\_type = 'images'

image\_shape = (80, 80, 3)

elif model in ['lstm', 'mlp']:

data\_type = 'features'

image\_shape = None

else:

raise ValueError("Invalid model. See train.py for options.")

start\_time = datetime.now()

predict(data\_type, seq\_length, saved\_model, image\_shape, video\_name, class\_limit)

time\_elapsed = datetime.now() - start\_time

print('Time elapsed (hh:mm:ss.ms) {}'.format(time\_elapsed))

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

main()

"""

Process an image that we can pass to our networks.

"""

from keras.preprocessing.image import img\_to\_array, load\_img

import numpy as np

def process\_image(image, target\_shape):

"""Given an image, process it and return the array."""

# Load the image.

h, w, \_ = target\_shape

image = load\_img(image, target\_size=(h, w))

# Turn it into numpy, normalize and return.

img\_arr = img\_to\_array(image)

x = (img\_arr / 255.).astype(np.float32)

return x

"""

Given a training log file, plot something.

"""

import csv

import matplotlib.pyplot as plt

def main(training\_log):

with open(training\_log) as fin:

reader = csv.reader(fin)

next(reader, None) # skip the header

accuracies = []

top\_5\_accuracies = []

cnn\_benchmark = [] # this is ridiculous

for epoch,acc,loss,top\_k\_categorical\_accuracy,val\_acc,val\_loss,val\_top\_k\_categorical\_accuracy in reader:

accuracies.append(float(val\_acc))

top\_5\_accuracies.append(float(val\_top\_k\_categorical\_accuracy))

cnn\_benchmark.append(0.65) # ridiculous

plt.plot(accuracies, label="accuracies")

plt.plot(top\_5\_accuracies, label="top\_5\_accuracies")

plt.plot(cnn\_benchmark, label="cnn\_benchmark")

plt.legend(loc="upper right")

plt.show()

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

#training\_log = 'data/logs/mlp-training-1489455559.7089438.log'

training\_log = 'data/logs/lstm-training-1540081899.9795177.log'

main(training\_log)

"""

Try to "classify" samples based on random chance and always guessing

the most popular category.

"""

import random

from data import DataSet

most\_pop = 'TennisSwing'

data = DataSet()

nb\_classes = len(data.classes)

print(nb\_classes)

# Try a random guess.

nb\_random\_matched = 0

nb\_mode\_matched = 0

for item in data.data:

choice = random.choice(data.classes)

actual = item[1]

if choice == actual:

nb\_random\_matched += 1

if actual == most\_pop:

nb\_mode\_matched += 1

random\_accuracy = nb\_random\_matched / len(data.data)

mode\_accuracy = nb\_mode\_matched / len(data.data)

print("Randomly matched %.2f%%" % (random\_accuracy \* 100))

print("Mode matched %.2f%%" % (mode\_accuracy \* 100))

"""

Train on images split into directories. This assumes we've split

our videos into frames and moved them to their respective folders.

Based on:

https://keras.io/preprocessing/image/

and

https://keras.io/applications/

"""

from keras.applications.inception\_v3 import InceptionV3

from keras.optimizers import SGD

from keras.preprocessing.image import ImageDataGenerator

from keras.models import Model

from keras.layers import Dense, GlobalAveragePooling2D

from keras.callbacks import ModelCheckpoint, TensorBoard, EarlyStopping

from data import DataSet

import os.path

data = DataSet()

# Helper: Save the model.

checkpointer = ModelCheckpoint(

filepath=os.path.join('data', 'checkpoints', 'inception.{epoch:03d}-{val\_loss:.2f}.hdf5'),

verbose=1,

save\_best\_only=True)

# Helper: Stop when we stop learning.

early\_stopper = EarlyStopping(patience=10)

# Helper: TensorBoard

tensorboard = TensorBoard(log\_dir=os.path.join('data', 'logs'))

def get\_generators():

train\_datagen = ImageDataGenerator(

rescale=1./255,

shear\_range=0.2,

horizontal\_flip=True,

rotation\_range=10.,

width\_shift\_range=0.2,

height\_shift\_range=0.2)

test\_datagen = ImageDataGenerator(rescale=1./255)

print(data, data.classes)

train\_generator = train\_datagen.flow\_from\_directory(

os.path.join('data', 'train'),

target\_size=(299, 299),

batch\_size=32,

classes=data.classes,

class\_mode='categorical')

validation\_generator = test\_datagen.flow\_from\_directory(

os.path.join('data', 'test'),

target\_size=(299, 299),

batch\_size=32,

classes=data.classes,

class\_mode='categorical')

return train\_generator, validation\_generator

def get\_model(weights='imagenet'):

# create the base pre-trained model

base\_model = InceptionV3(weights=weights, include\_top=False)

# add a global spatial average pooling layer

x = base\_model.output

x = GlobalAveragePooling2D()(x)

# let's add a fully-connected layer

x = Dense(1024, activation='relu')(x)

# and a logistic layer

predictions = Dense(len(data.classes), activation='softmax')(x)

# this is the model we will train

model = Model(inputs=base\_model.input, outputs=predictions)

print(model.summary())

return model

def freeze\_all\_but\_top(model):

"""Used to train just the top layers of the model."""

# first: train only the top layers (which were randomly initialized)

# i.e. freeze all convolutional InceptionV3 layers

for layer in model.layers[:-2]:

layer.trainable = False

# compile the model (should be done \*after\* setting layers to non-trainable)

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy'])

return model

def freeze\_all\_but\_mid\_and\_top(model):

"""After we fine-tune the dense layers, train deeper."""

# we chose to train the top 2 inception blocks, i.e. we will freeze

# the first 172 layers and unfreeze the rest:

for layer in model.layers[:172]:

layer.trainable = False

for layer in model.layers[172:]:

layer.trainable = True

# we need to recompile the model for these modifications to take effect

# we use SGD with a low learning rate

model.compile(

optimizer=SGD(lr=0.0001, momentum=0.9),

loss='categorical\_crossentropy',

metrics=['accuracy', 'top\_k\_categorical\_accuracy'])

return model

def train\_model(model, nb\_epoch, generators, callbacks=[]):

train\_generator, validation\_generator = generators

model.fit\_generator(

train\_generator,

steps\_per\_epoch=100,

validation\_data=validation\_generator,

validation\_steps=10,

epochs=nb\_epoch,

callbacks=callbacks)

return model

def main(weights\_file):

model = get\_model()

generators = get\_generators()

if weights\_file is None:

print("Loading network from ImageNet weights.")

# Get and train the top layers.

model = freeze\_all\_but\_top(model)

model = train\_model(model, 10, generators)

else:

print("Loading saved model: %s." % weights\_file)

model.load\_weights(weights\_file)

# Get and train the mid layers.

model = freeze\_all\_but\_mid\_and\_top(model)

model = train\_model(model, 1000, generators,

[checkpointer, early\_stopper, tensorboard])

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

weights\_file = None

main(weights\_file)

"""

Train our RNN on extracted features or images.

"""

from keras.callbacks import TensorBoard, ModelCheckpoint, EarlyStopping, CSVLogger

from models import ResearchModels

from data import DataSet

import time

import os.path

def train(data\_type, seq\_length, model, saved\_model=None,

class\_limit=None, image\_shape=None,

load\_to\_memory=False, batch\_size=32, nb\_epoch=100):

# Helper: Save the model.

checkpointer = ModelCheckpoint(

filepath=os.path.join('data', 'checkpoints', model + '-' + data\_type + \

'.{epoch:03d}-{val\_loss:.3f}.hdf5'),

verbose=1,

save\_best\_only=True)

# Helper: TensorBoard

tb = TensorBoard(log\_dir=os.path.join('data', 'logs', model))

# Helper: Stop when we stop learning.

early\_stopper = EarlyStopping(patience=5)

# Helper: Save results.

timestamp = time.time()

csv\_logger = CSVLogger(os.path.join('data', 'logs', model + '-' + 'training-' + \

str(timestamp) + '.log'))

# Get the data and process it.

if image\_shape is None:

data = DataSet(

seq\_length=seq\_length,

class\_limit=class\_limit

)

else:

data = DataSet(

seq\_length=seq\_length,

class\_limit=class\_limit,

image\_shape=image\_shape

)

# Get samples per epoch.

# Multiply by 0.7 to attempt to guess how much of data.data is the train set.

steps\_per\_epoch = (len(data.data) \* 0.6) // batch\_size

if load\_to\_memory:

# Get data.

X, y = data.get\_all\_sequences\_in\_memory('train', data\_type)

X\_test, y\_test = data.get\_all\_sequences\_in\_memory('test', data\_type)

else:

# Get generators.

generator = data.frame\_generator(batch\_size, 'train', data\_type)

val\_generator = data.frame\_generator(batch\_size, 'test', data\_type)

# Get the model.

rm = ResearchModels(len(data.classes), model, seq\_length, saved\_model)

# Fit!

if load\_to\_memory:

# Use standard fit.

rm.model.fit(

X,

y,

batch\_size=batch\_size,

validation\_data=(X\_test, y\_test),

verbose=1,

callbacks=[tb, early\_stopper, csv\_logger],

epochs=nb\_epoch)

else:

# Use fit generator.

rm.model.fit\_generator(

generator=generator,

steps\_per\_epoch=steps\_per\_epoch,

epochs=nb\_epoch,

verbose=1,

callbacks=[tb, early\_stopper, csv\_logger, checkpointer],

validation\_data=val\_generator,

validation\_steps=40,

workers=4)

print(rm.model.summary())

def main():

"""These are the main training settings. Set each before running

this file."""

# model can be one of lstm, lrcn, mlp, conv\_3d, c3d

model = 'lstm'

saved\_model = None # None or weights file

class\_limit = 63 # int, can be 1-101 or None

seq\_length = 40

load\_to\_memory = False # pre-load the sequences into memory

batch\_size = 2

nb\_epoch = 1000

# Chose images or features and image shape based on network.

if model in ['conv\_3d', 'c3d', 'lrcn']:

data\_type = 'images'

image\_shape = (80, 80, 3)

elif model in ['lstm', 'mlp']:

data\_type = 'features'

image\_shape = None

else:

raise ValueError("Invalid model. See train.py for options.")

train(data\_type, seq\_length, model, saved\_model=saved\_model,

class\_limit=class\_limit, image\_shape=image\_shape,

load\_to\_memory=load\_to\_memory, batch\_size=batch\_size, nb\_epoch=nb\_epoch)

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

main()

"""

Classify a few images through our CNN.

"""

import numpy as np

import operator

import random

import glob

import os.path

from data import DataSet

from processor import process\_image

from keras.models import load\_model

def main(nb\_images=5):

"""Spot-check `nb\_images` images."""

data = DataSet()

#model = load\_model('data/checkpoints/inception.057-1.16.hdf5')

model = load\_model('data/checkpoints/inception.016-1.46.hdf5')

# Get all our test images.

images = glob.glob(os.path.join('data', 'test', '\*\*', '\*.jpg'))

for \_ in range(nb\_images):

print('-'\*80)

# Get a random row.

sample = random.randint(0, len(images) - 1)

image = images[sample]

# Turn the image into an array.

print(image)

image\_arr = process\_image(image, (299, 299, 3))

image\_arr = np.expand\_dims(image\_arr, axis=0)

# Predict.

predictions = model.predict(image\_arr)

# Show how much we think it's each one.

label\_predictions = {}

for i, label in enumerate(data.classes):

label\_predictions[label] = predictions[0][i]

sorted\_lps = sorted(label\_predictions.items(), key=operator.itemgetter(1), reverse=True)

for i, class\_prediction in enumerate(sorted\_lps):

# Just get the top five.

if i > 4:

break

print("%s: %.2f" % (class\_prediction[0], class\_prediction[1]))

i += 1

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

main()

"""

Validate our RNN. Basically just runs a validation generator on

about the same number of videos as we have in our test set.

"""

from keras.callbacks import TensorBoard, ModelCheckpoint, CSVLogger

from models import ResearchModels

from data import DataSet

def validate(data\_type, model, seq\_length=40, saved\_model=None,

class\_limit=None, image\_shape=None):

batch\_size = 32

# Get the data and process it.

if image\_shape is None:

data = DataSet(

seq\_length=seq\_length,

class\_limit=class\_limit

)

else:

data = DataSet(

seq\_length=seq\_length,

class\_limit=class\_limit,

image\_shape=image\_shape

)

val\_generator = data.frame\_generator(batch\_size, 'test', data\_type)

# Get the model.

rm = ResearchModels(len(data.classes), model, seq\_length, saved\_model)

# Evaluate!

results = rm.model.evaluate\_generator(

generator=val\_generator,

val\_samples=3200)

print(results)

print(rm.model.metrics\_names)

def main():

model = 'lstm'

#saved\_model = 'data/checkpoints/lstm-features.026-0.239.hdf5'

saved\_model = 'data/checkpoints/lstm-features.017-1.156.hdf5'

if model == 'conv\_3d' or model == 'lrcn':

data\_type = 'images'

image\_shape = (80, 80, 3)

else:

data\_type = 'features'

image\_shape = None

validate(data\_type, model, saved\_model=saved\_model,

image\_shape=image\_shape, class\_limit=4)

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

main()