**Imports**

**# importing libraries**

%matplotlib inline

import cv2

import os

import numpy as np

import keras

import matplotlib.pyplot as plt

import download

from random import shuffle

from keras.applications import VGG16

from keras import backend as K

from keras.models import Model, Sequential

from keras.layers import Input

from keras.layers import LSTM

from keras.layers import Dense, Activation

import sys

import h5py

# keras version

keras.\_\_version\_\_

**Helper Functions**

We will use the function ```print\_progress``` to print the amount of videos processed and ```download\_data``` to download the datasets

# count, max\_count

#

def print\_progress(count, max\_count):

# Percentage completion.

pct\_complete = count / max\_count

msg = "\r- Progress: {0:.1%}".format(pct\_complete)

# Print it.

sys.stdout.write(msg)

sys.stdout.flush()

def download\_data(in\_dir, url):

if not os.path.exists(in\_dir):

os.makedirs(in\_dir)

download.maybe\_download\_and\_extract(url,in\_dir)

**Load Data**

Firstly, we define the directory to place the video dataset

# defined directory

in\_dir = "data"

# dataset downlodable link

url\_hockey = "http://visilab.etsii.uclm.es/personas/oscar/FightDetection/HockeyFights.zip"

to download the dataset and decompress it:

#download\_data(in\_dir,url\_hockey)

Copy some of the data-dimensions for convenience.

# Frame size

img\_size = 224

img\_size\_touple = (img\_size, img\_size)

# Number of channels (RGB)

num\_channels = 3

# Flat frame size

img\_size\_flat = img\_size \* img\_size \* num\_channels

# Number of classes for classification (Violence-No Violence)

num\_classes = 2

# Number of files to train

\_num\_files\_train = 1

# Number of frames per video

\_images\_per\_file = 20

# Number of frames per training set

\_num\_images\_train = \_num\_files\_train \* \_images\_per\_file

# Video extension

video\_exts = ".avi"

**Helper-function for getting video frames**

Function used to get 20 frames from a video file and convert the frame to a suitable format for the neural net.

def get\_frames(current\_dir, file\_name):

in\_file = os.path.join(current\_dir, file\_name)

images = []

vidcap = cv2.VideoCapture(in\_file)

success,image = vidcap.read()

count = 0

while count<\_images\_per\_file:

RGB\_img = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

res = cv2.resize(RGB\_img, dsize=(img\_size, img\_size),

interpolation=cv2.INTER\_CUBIC)

images.append(res)

success,image = vidcap.read()

count += 1

resul = np.array(images)

resul = (resul / 255.).astype(np.float16)

return resul

**Helper function to get the names of the data downloaded and label it**

def label\_video\_names(in\_dir):

# list containing video names

names = []

# list containin video labels [1, 0] if it has violence and [0, 1] if not

labels = []

for current\_dir, dir\_names,file\_names in os.walk(in\_dir):

for file\_name in file\_names:

if file\_name[0:2] == 'fi':

labels.append([1,0])

names.append(file\_name)

elif file\_name[0:2] == 'no':

labels.append([0,1])

names.append(file\_name)

c = list(zip(names,labels))

# Suffle the data (names and labels)

shuffle(c)

names, labels = zip(\*c)

return names, labels

**Plot a video frame to see if data is correct**

# First get the names and labels of the whole videos

names, labels = label\_video\_names(in\_dir)

Then we are going to load 20 frames of one video, for example

names[12]

The video has violence, look at the name of the video, starts with 'fi'

frames = get\_frames(in\_dir, names[12])

Convert back the frames to uint8 pixel format to plot the frame

visible\_frame = (frames\*255).astype('uint8')

plt.imshow(visible\_frame[3])

plt.imshow(visible\_frame[15])

**Pre-Trained Model: VGG16**

The following creates an instance of the pre-trained VGG16 model using the Keras API. This automatically downloads the required files if you don't have them already.

The VGG16 model contains a convolutional part and a fully-connected (or dense) part which is used for classification. If include\_top=True then the whole VGG16 model is downloaded which is about 528 MB. If include\_top=False then only the convolutional part of the VGG16 model is downloaded which is just 57 MB.

image\_model = VGG16(include\_top=True, weights='imagenet')

Let's see the model summary

image\_model.summary()

We can observe the shape of the tensors expected as input by the pre-trained VGG16 model. In this case it is images of shape 224 x 224 x 3. Note that we have defined the frame size as 224x224x3. The video frame will be the input of the VGG16 net.

input\_shape = image\_model.layers[0].output\_shape[1:3]

input\_shape

**VGG16 model flowchart**

The following chart shows how the data flows when using the VGG16 model for Transfer Learning. First we input and process 20 video frames in batch with the VGG16 model. Just prior to the final classification layer of the VGG16 model, we save the so-called Transfer Values to a cache-file.

The reason for using a cache-file is that it takes a long time to process an image with the VGG16 model. If each image is processed more than once then we can save a lot of time by caching the transfer-values.

When all the videos have been processed through the VGG16 model and the resulting transfer-values saved to a cache file, then we can use those transfer-values as the input to LSTM neural network. We will then train the second neural network using the classes from the violence dataset (Violence, No-Violence), so the network learns how to classify images based on the transfer-values from the VGG16 model.

# We will use the output of the layer prior to the final

# classification-layer which is named fc2. This is a fully-connected (or dense) layer.

transfer\_layer = image\_model.get\_layer('fc2')

image\_model\_transfer = Model(inputs=image\_model.input,

outputs=transfer\_layer.output)

transfer\_values\_size = K.int\_shape(transfer\_layer.output)[1]

print("The input of the VGG16 net have dimensions:",K.int\_shape(image\_model.input)[1:3])

print("The output of the selecter layer of VGG16 net have dimensions: ", transfer\_values\_size)

**Function to process 20 video frames through VGG16 and get transfer values**

def get\_transfer\_values(current\_dir, file\_name):

# Pre-allocate input-batch-array for images.

shape = (\_images\_per\_file,) + img\_size\_touple + (3,)

image\_batch = np.zeros(shape=shape, dtype=np.float16)

image\_batch = get\_frames(current\_dir, file\_name)

# Pre-allocate output-array for transfer-values.

# Note that we use 16-bit floating-points to save memory.

shape = (\_images\_per\_file, transfer\_values\_size)

transfer\_values = np.zeros(shape=shape, dtype=np.float16)

transfer\_values = \

image\_model\_transfer.predict(image\_batch)

return transfer\_values

**Generator that process one video through VGG16 each function call**

def proces\_transfer(vid\_names, in\_dir, labels):

count = 0

tam = len(vid\_names)

# Pre-allocate input-batch-array for images.

shape = (\_images\_per\_file,) + img\_size\_touple + (3,)

while count<tam:

video\_name = vid\_names[count]

image\_batch = np.zeros(shape=shape, dtype=np.float16)

image\_batch = get\_frames(in\_dir, video\_name)

# Note that we use 16-bit floating-points to save memory.

shape = (\_images\_per\_file, transfer\_values\_size)

transfer\_values = np.zeros(shape=shape, dtype=np.float16)

transfer\_values = \

image\_model\_transfer.predict(image\_batch)

labels1 = labels[count]

aux = np.ones([20,2])

labelss = labels1\*aux

yield transfer\_values, labelss

count+=1

**Functions to save transfer values from VGG16 to later use**

We are going to define functions to get the transfer values from VGG16 with defined number of files. Then save the transfer values files used from training in one file and the ones uses for testing in another one.

def make\_files(n\_files):

gen = proces\_transfer(names\_training, in\_dir, labels\_training)

numer = 1

# Read the first chunk to get the column dtypes

chunk = next(gen)

row\_count = chunk[0].shape[0]

row\_count2 = chunk[1].shape[0]

with h5py.File('prueba.h5', 'w') as f:

# Initialize a resizable dataset to hold the output

maxshape = (None,) + chunk[0].shape[1:]

maxshape2 = (None,) + chunk[1].shape[1:]

dset = f.create\_dataset('data', shape=chunk[0].shape, maxshape=maxshape,

chunks=chunk[0].shape, dtype=chunk[0].dtype)

dset2 = f.create\_dataset('labels', shape=chunk[1].shape, maxshape=maxshape2,

chunks=chunk[1].shape, dtype=chunk[1].dtype)

# Write the first chunk of rows

dset[:] = chunk[0]

dset2[:] = chunk[1]

for chunk in gen:

if numer == n\_files:

break

# Resize the dataset to accommodate the next chunk of rows

dset.resize(row\_count + chunk[0].shape[0], axis=0)

dset2.resize(row\_count2 + chunk[1].shape[0], axis=0)

# Write the next chunk

dset[row\_count:] = chunk[0]

dset2[row\_count:] = chunk[1]

# Increment the row count

row\_count += chunk[0].shape[0]

row\_count2 += chunk[1].shape[0]

print\_progress(numer, n\_files)

numer += 1

def make\_files\_test(n\_files):

gen = proces\_transfer(names\_test, in\_dir, labels\_test)

numer = 1

# Read the first chunk to get the column dtypes

chunk = next(gen)

row\_count = chunk[0].shape[0]

row\_count2 = chunk[1].shape[0]

with h5py.File('pruebavalidation.h5', 'w') as f:

# Initialize a resizable dataset to hold the output

maxshape = (None,) + chunk[0].shape[1:]

maxshape2 = (None,) + chunk[1].shape[1:]

dset = f.create\_dataset('data', shape=chunk[0].shape, maxshape=maxshape,

chunks=chunk[0].shape, dtype=chunk[0].dtype)

dset2 = f.create\_dataset('labels', shape=chunk[1].shape, maxshape=maxshape2,

chunks=chunk[1].shape, dtype=chunk[1].dtype)

# Write the first chunk of rows

dset[:] = chunk[0]

dset2[:] = chunk[1]

for chunk in gen:

if numer == n\_files:

break

# Resize the dataset to accommodate the next chunk of rows

dset.resize(row\_count + chunk[0].shape[0], axis=0)

dset2.resize(row\_count2 + chunk[1].shape[0], axis=0)

# Write the next chunk

dset[row\_count:] = chunk[0]

dset2[row\_count:] = chunk[1]

# Increment the row count

row\_count += chunk[0].shape[0]

row\_count2 += chunk[1].shape[0]

print\_progress(numer, n\_files)

numer += 1

**Split the dataset into training set and test set**

We are going to split the dataset into training set and testing. The training set is used to train the model and the test set to check the model accuracy.

training\_set = int(len(names)\*0.8)

test\_set = int(len(names)\*0.2)

names\_training = names[0:training\_set]

names\_test = names[training\_set:]

labels\_training = labels[0:training\_set]

labels\_test = labels[training\_set:]

Then we are going to process all video frames through VGG16 and save the transfer values.

make\_files(training\_set)

make\_files\_test(test\_set)

**Load the cached transfer values into memory**

We have already saved all the videos transfer values into disk. But we have to load those transfer values into memory in order to train the LSTM net. One question would be: why not process transfer values and load them into RAM memory? Yes is a more eficient way to train the second net. But if you have to train the LSTM in different ways in order to see which way gets the best accuracy, if you didn't save the transfer values into disk you would have to process the whole videos each training. It's very time consuming processing the videos through VGG16 net.

In order to load the saved transfer values into RAM memory we are going to use this two functions:

def process\_alldata\_training():

joint\_transfer=[]

frames\_num=20

count = 0

with h5py.File('prueba.h5', 'r') as f:

X\_batch = f['data'][:]

y\_batch = f['labels'][:]

for i in range(int(len(X\_batch)/frames\_num)):

inc = count+frames\_num

joint\_transfer.append([X\_batch[count:inc],y\_batch[count]])

count =inc

data =[]

target=[]

for i in joint\_transfer:

data.append(i[0])

target.append(np.array(i[1]))

return data, target

def process\_alldata\_test():

joint\_transfer=[]

frames\_num=20

count = 0

with h5py.File('pruebavalidation.h5', 'r') as f:

X\_batch = f['data'][:]

y\_batch = f['labels'][:]

for i in range(int(len(X\_batch)/frames\_num)):

inc = count+frames\_num

joint\_transfer.append([X\_batch[count:inc],y\_batch[count]])

count =inc

data =[]

target=[]

for i in joint\_transfer:

data.append(i[0])

target.append(np.array(i[1]))

return data, target

data, target = process\_alldata\_training()

data\_test, target\_test = process\_alldata\_test()

**Recurrent Neural Network**

The basic building block in a Recurrent Neural Network (RNN) is a Recurrent Unit (RU). There are many different variants of recurrent units such as the rather clunky LSTM (Long-Short-Term-Memory) and the somewhat simpler GRU (Gated Recurrent Unit) which we will use in this tutorial. Experiments in the literature suggest that the LSTM and GRU have roughly similar performance. Even simpler variants also exist and the literature suggests that they may perform even better than both LSTM and GRU, but they are not implemented in Keras which we will use in this tutorial.

A recurrent neuron has an internal state that is being updated every time the unit receives a new input. This internal state serves as a kind of memory. However, it is not a traditional kind of computer memory which stores bits that are either on or off. Instead the recurrent unit stores floating-point values in its memory-state, which are read and written using matrix-operations so the operations are all differentiable. This means the memory-state can store arbitrary floating-point values (although typically limited between -1.0 and 1.0) and the network can be trained like a normal neural network using Gradient Descent.

**Define LSTM architecture**

When defining the LSTM architecture we have to take into account the dimensions of the transfer values. From each frame the VGG16 network obtains as output a vector of 4096 transfer values. From each video we are processing 20 frames so we will have 20 x 4096 values per video. The classification must be done taking into account the 20 frames of the video. If any of them detects violence, the video will be classified as violent.

The first input dimension of LSTM neurons is the temporal dimension, in our case it is 20. The second is the size of the features vector (transfer values).

chunk\_size = 4096

n\_chunks = 20

rnn\_size = 512

model = Sequential()

model.add(LSTM(rnn\_size, input\_shape=(n\_chunks, chunk\_size)))

model.add(Dense(1024))

model.add(Activation('relu'))

model.add(Dense(50))

model.add(Activation('sigmoid'))

model.add(Dense(2))

model.add(Activation('softmax'))

model.compile(loss='mean\_squared\_error', optimizer='adam',metrics=['accuracy'])

**Model training**

epoch = 5

batchS = 10

history = model.fit(np.array(data[0:750]), np.array(target[0:750]), epochs=epoch,

validation\_data=(np.array(data[750:]), np.array(target[750:])),

batch\_size=batchS, verbose=2)

**Test the model**

We are going to test the model with 20 % of the total videos. This videos have not been used to train the network.

result = model.evaluate(np.array(data\_test), np.array(target\_test))

**Print the model accuracy**

for name, value in zip(model.metrics\_names, result):

print(name, value)

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

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

plt.title('model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

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

plt.savefig('destination\_path.eps', format='eps', dpi=1000)

plt.show()

# 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', 'validation'], loc='upper left')

plt.savefig('destination\_path1.eps', format='eps', dpi=1000)

plt.show()