Skateboarding trick classifier

|  |
| --- |
| import tensorflow as tf; import os; import cv2; import numpy as np; import tqdm; from sklearn.preprocessing import LabelBinarizer;  BASE\_PATH = 'D:/SkateboardML/Tricks'  VIDEOS\_PATH = os.path.join(BASE\_PATH, '\*\*','\*.mov') SEQUENCE\_LENGTH = 40 |

To start with this program you need to import the following modules from python. You will use tensorflow to build the model with its keras wrapper class. You will use the os module to find some of the video paths. You use cv2 to read in the images for processing. You will finally use tqdm to make a count the number of iterations the loop makes it through. We will also use LabelBinarizer to let us select which trick it is.

|  |
| --- |
| def frame\_generator():  video\_paths = tf.io.gfile.glob(VIDEOS\_PATH)  np.random.shuffle(video\_paths)  for video\_path in video\_paths:  #frames = []  cap = cv2.VideoCapture(video\_path)  num\_frames = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))  sample\_every\_frame = max(1, num\_frames // SEQUENCE\_LENGTH)  current\_frame = 0   #label = os.path.basename(os.path.dirname(video\_path))   max\_images = SEQUENCE\_LENGTH  while True:  success, frame = cap.read()  if not success:  break   if current\_frame % sample\_every\_frame == 0:  # OPENCV reads in BGR, tensorflow expects RGB so we invert the order  frame = frame[:, :, ::-1]  img = tf.image.resize(frame, (299, 299))  img = tf.keras.applications.inception\_v3.preprocess\_input(  img)  max\_images -= 1  yield img, video\_path   if max\_images == 0:  break  current\_frame += 1  # `from\_generator` might throw a warning, expected to disappear in upcoming versions: # https://www.tensorflow.org/versions/r2.0/api\_docs/python/tf/data/Dataset#for\_example\_2 dataset = tf.data.Dataset.from\_generator(frame\_generator,  output\_types=(tf.float32, tf.string),  output\_shapes=((299, 299, 3), ()))  dataset = dataset.batch(2).prefetch(tf.data.experimental.AUTOTUNE) |

Starting from the dataset variable we call the tf.data.Dataset.from\_generator function to generate the frames that are to be featured by the CNN. If we exam the frame\_generator function, when we first arrive in the function we first start by assigning the VIDEOS\_PATH variable to a different variable using tf.io.gfile.glob() function then shuffle that list so that we are always choosing a different video to feature extract from. This is basically just preparing the data by converting it from bgr to rgb because opencv reads in bgr. Then we resize the image to (299,299,3) because we are going to input the images into a pretrained model that accepts images of that size. Then we preprocess the image weights so that they are changed into the range of (-1,1) to fit into the pretrained model.we do this as a max of 40 times per image. If there is a video of 160 frames it'll read every 4th frame. Then we prepare the data in the dataset variables with expected output size and the string it is supposed to come with.

|  |
| --- |
| inception\_v3 = tf.keras.applications.InceptionV3(include\_top=False, weights='imagenet')  x = inception\_v3.output  # We add Average Pooling to transform the feature map from # 8 \* 8 \* 2048 to 1 x 2048, as we don't need spatial information pooling\_output = tf.keras.layers.GlobalAveragePooling2D()(x)  feature\_extraction\_model = tf.keras.Model(inception\_v3.input, pooling\_output) |

Then we setup the feature extraction model. In the first line we call the pretrained model object from tensorflow. We then get the output size and assign it to an arbitrary value to be flattened because we do not need spatial information. So we flatten the array using the Global average pooling function. We then finish setting up the models input and expected output in the final line.

|  |
| --- |
| current\_path = None all\_features = []  for img, batch\_paths in tqdm.tqdm(dataset):  batch\_features = feature\_extraction\_model(img)  batch\_features = tf.reshape(batch\_features, (batch\_features.shape[0], -1))    for features, path in zip(batch\_features.numpy(), batch\_paths.numpy()):  if path != current\_path and current\_path is not None:  output\_path = current\_path.decode().replace('.mov', '.npy')  np.save(output\_path, all\_features)  all\_features = []    current\_path = path  all\_features.append(features) |

Then we run this loop. What this loop does is that it iterates through all the generated frames and paths and so this loop extract key features from these list of paths of 40 and images then it stores each images features are stored in a 1x2048 array there are 40 images per video so your basically left with a 40x2048 npy file.

|  |
| --- |
| LABELS = ['Ollie','Kickflip','Shuvit']  encoder = LabelBinarizer() encoder.fit(LABELS)  model = tf.keras.Sequential([  tf.keras.layers.Masking(mask\_value=0.),  tf.keras.layers.LSTM(512, dropout=0.5, recurrent\_dropout=0.5),  tf.keras.layers.Dense(256, activation='relu'),  tf.keras.layers.Dropout(0.5),  tf.keras.layers.Dense(len(LABELS), activation='softmax') ]) |

We create our labels so that the Neural net can choose between the 3 tricks. Then we create a new variable called encoder so that we can binarize the 3 options we have and we then use the fit function to fit the encoder to the tricks or classes we want to be classified. We then move on to defining our model. The model that we define basically takes our input and gives us our output. We are building our model with the mind that we have one input and one output so we use the Sequential object in keras to reflect that. We then define the layers of our sequential model. Each layer does something different for our model. The Masking layer ignores padding so that it makes the learning process more efficient because all elements of our array will not always be filled with unique values. The next layer we define is our LSTM layer which is a derivative of a Recurrent Neural Network. A RNN is… The arguments that you can provide are numerous but the ones that we use for our LSTM application are the units, the dropout, and the recurrent dropout. We set the units of neurons to 512. I choose this number because it is smaller than 2048 which is the length of the array. If we increased the number of neurons then we would make the network more powerful but then we would also increase train time to some degree. The dropout argument we use is probabilistically excluded from activation and weight updates while training a network. This has the effect of reducing overfitting and improving model performance. Recurrent dropout masks (or "drops") the connections between the recurrent units. We then go on to define our dense layer. The dense layer defines how many neurons we have for our application so we have 256 neurons available. We also provide an activation argument with it. We then move on to our own independent Dropout layer which we just use as to not overfit the data. We then move on to our last dense layer Which is just set to the length of our labels variable so 3 because we have three different options for probabilistic output.

|  |
| --- |
| model.compile(loss='categorical\_crossentropy',  optimizer='rmsprop',  metrics=['accuracy', 'top\_k\_categorical\_accuracy']) |

we then go on to use the compile function. The compile function is used to configure your model with losses and other metrics. The loss function calculates a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. We choose this loss function specifically because we are doing multiclass classification. Then we set our optimizer which is basically our learning rate. Our learning rate will determine how long it takes our machine learning algorithm to converge in accuracy between the test dataset and the training set. Then finally we have our metrics which just gives us how often the prediction matches its label and how often the predictions are in the top k of categories.

|  |
| --- |
| with open('testlist02.txt') as f:  test\_list = [row.strip() for row in list(f)]  with open('trainlist02.txt') as f:  train\_list = [row.strip() for row in list(f)]  train\_list = [row.split(' ')[0] for row in train\_list] |

Here we are just populating our arrays with data from our test train list.

|  |
| --- |
| def make\_generator(file\_list):  def generator():  np.random.shuffle(file\_list)  for path in file\_list:  full\_path = os.path.join(BASE\_PATH + '/', path).replace('.mov', '.npy')   label = os.path.basename(os.path.dirname(path))  features = np.load(full\_path)   padded\_sequence = np.zeros((SEQUENCE\_LENGTH, 2048))  padded\_sequence[0:len(features)] = np.array(features)   transformed\_label = encoder.transform([label])  yield padded\_sequence, transformed\_label[0]  return generator  train\_dataset = tf.data.Dataset.from\_generator(make\_generator(train\_list),  output\_types=(tf.float32, tf.int16),  output\_shapes=((SEQUENCE\_LENGTH, 2048), (len(LABELS)))) train\_dataset = train\_dataset.batch(16).prefetch(tf.data.experimental.AUTOTUNE)   valid\_dataset = tf.data.Dataset.from\_generator(make\_generator(test\_list),  output\_types=(tf.float32, tf.int16),  output\_shapes=((SEQUENCE\_LENGTH, 2048), (len(LABELS)))) valid\_dataset = valid\_dataset.batch(16).prefetch(tf.data.experimental.AUTOTUNE) |

In these lines of code we have seemingly a lot going on. We have a generator inside a function. We also have two variables being declared and assigned however, this is basically just like the same code above except slightly different. In this instance we are just padding the 40,2048 array with zeros where needed it also transforms the label. From then on everything else is the same from the feature extraction phase.

|  |
| --- |
| tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir='log', update\_freq=1000) model.fit(train\_dataset, epochs=17, callbacks=[tensorboard\_callback], validation\_data=valid\_dataset) |

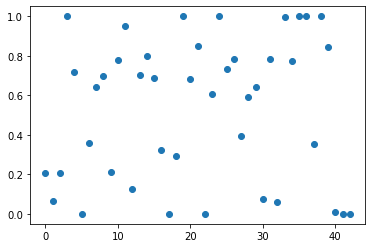
TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more. Then we are just left with the last model fit function. Which just starts the training phase.

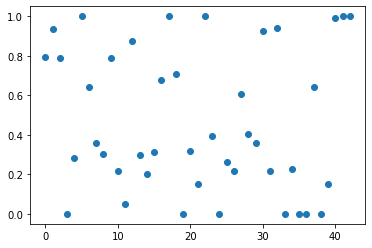
Interpreting the data

After we run all the code above there is a lot that we can do and show after compiling all the data we can use model.predict(valid\_dataset) to get a view of the predicted outcomes of the test\_list. When run we are met with a 2d numpy array with all of our predicted values which can be stored to be then plotted. For example we can store the model.predict(valid\_dataset) outcome in some variable then import matplotlib.pyplot as plt. With plt we can plot the predictions. An example would be:

|  |
| --- |
| import matplotlib.pyplot as plt  predict = model.predict(valid\_dataset) Kickflip = predict[:,0] x = np.arange(len(Kickflip)) plt.scatter(x, Kickflip) |

Which would yield a thing like this scatter plot that looks like below

for kickflips and

for ollie. If we look at the data points we can see that the data is really sporadic. If we count the ollie plot only 13 plot points are above 0.8 probability so let's find out why the model is so certain on these points and so uncertain on other points.

<https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/#:~:text=Dropout%20is%20a%20regularization%20method,overfitting%20and%20improving%20model%20performance.>

<https://stackoverflow.com/questions/44924690/keras-the-difference-between-lstm-dropout-and-lstm-recurrent-dropout>

<https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>

<https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d>

<https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>

<https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2>

<https://www.tensorflow.org/tensorboard/get_started>