Zohreh Shaghaghian **Task: predicting Pushing**

I proceeded two methods for this task. 1. CNN and ,2. CNN+LSTM. Both models and results are elaborated in this report.

Data preparation:

I found around 30 short videos for pushing online and I created around 50 short videos in which I am pushing some object. About the same number of videos are used for other actions such as walking, archery, jumping, etc. In order to make the labeling process easier, I separated all pushing videos from all other action videos. In total I got around 130 videos for pushing and 100 videos of other actions

Since videos were separated, I rename them automatically using following code:

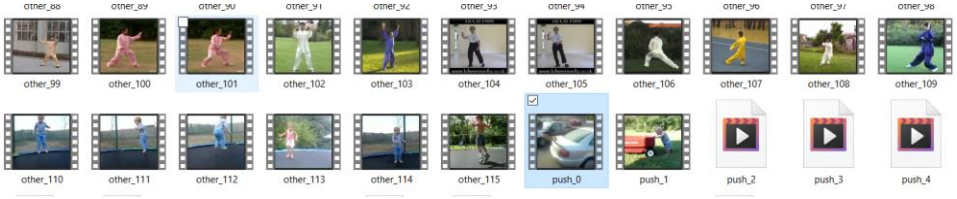
```
# Renaming video files in directory to make labeling easy

def rename_file (path , name):
    i=0
    for filename in os.listdir(path):
        os.rename(os.path.join(path,filename), os.path.join(path, name + str(i)+'.avi'))
        i = i +1

all_video_push = "D:\\tamu\\courses\\DeepLearning\\ProjectPart4\\pushing"
all_video_other = "D:\\tamu\\courses\\DeepLearning\\ProjectPart4\\other"

rename_file(all_video_push, 'push_')
rename_file(all_video_other, 'other_')
```

After each video is renamed, I pasted all 130 videos in one folder:



Getting video frames and saving frames in image directory.

```
def get_frames(video_path, dest_path):
    file_name = video_path.split('.')[0].split('\\')[-1]
    vidcap = cv2.VideoCapture(video_path)

    sec = 0
    frameRate = 0.5 #//it will capture image in each 0.5 second
    count = 1
    hasFrames = True
    while hasFrames:
        vidcap.set(cv2.CAP_PROP_POS_MSEC,sec*1000)
        hasFrames,image = vidcap.read()
        if hasFrames:
            cv2.imwrite(dest_path+'\\'+file_name+'_'+ str(count)+".jpg", image)     # save frame as JPG file

        count = count + 1
        sec = sec + frameRate
        sec = round(sec, 2)
```

```
# getting captions from videos and put them in related derectories:

video_dir = "D:\\tamu\\courses\\DeepLearning\\ProjectPart4\\other"
img_dir = "D:\\tamu\\courses\\DeepLearning\\ProjectPart4\\other_img"

for i in range(len(os.listdir(video_dir))):
    video_path = video_dir + '\\' + os.listdir(video_dir)[i]
    dest_path = img_dir

    get_frames(video_path, dest_path)
```

Note: I could manage to make 40 of my pushing videos openposed.

Getting labels:

Since the names of the videos represent the labels of the frames, with this method I did not need manual labeling.

```
train_label =[]
for file in os.listdir(img_dir):
    label = file.split('_')[0]
    if label == 'other':
        train_label.append(0)
    else:
        train_label.append(1)
print (len(train_label))
```

Resizing the images:

```
def get_Xtrain(file_name):
    X_train = []
    for i , filename in enumerate(glob.glob(file_name)):
        image = cv2.imread(filename)
        res_img = cv2.resize(image, dsize=(250,250), interpolation=cv2.INTER_CUBIC)
        X_train.append(res_img)

    return X_train

X = get_Xtrain(img_dir + '\\*.jpg')
```

Getting numpy array of images so they become appropriate tensors for model network.

Also the values are normalized since as we know Neural Network does not work well with big values.

```
#transfering the train_list and label_list to array
X = np.asarray(X)
y = np.asarray(train_label)
```

```
print('X.shape: ', X.shape)
print('y.shape: ', y.shape)
```

x.shape: (3170, 250,250,3)
y.shape: (3170,)

From Here Two methods are followed:

1. **Method 1: CNN + transfer learning using VGG 16**

The first method does not consider timeseries and only get the feature of frames one by one.
Splitting data to train, and validation. (70% for training and 30% for validation)
(I used all videos for training, and I prepared 5 more videos later for testing)

```
## split data to train and validation
from sklearn.model_selection import train_test_split

x_train, x_valid, y_train, y_valid = train_test_split(X, y, test_size=0.3, random_state=42)  #by default shuffle is true
```

Model Configuration

Using pre-trained model VGG16:

```
from keras.applications import VGG16

conv_base = VGG16(weights= 'imagenet', include_top=False, input_shape=(x_train.shape[1], x_train.shape[2], 3))
```

Two ways to proceed:

1. Running the conv_base over our dataset, record the out put as numpy array and use this data as input to a standalone densely connected classifier. ==> fast and cheap, but does not allow data augmentation

2. extend the conv_base by adding Dense layers on top, run the whole model end to end ==> far more expensive, But allows data augmentation

I use the second method where I do data Augmentation for x_train

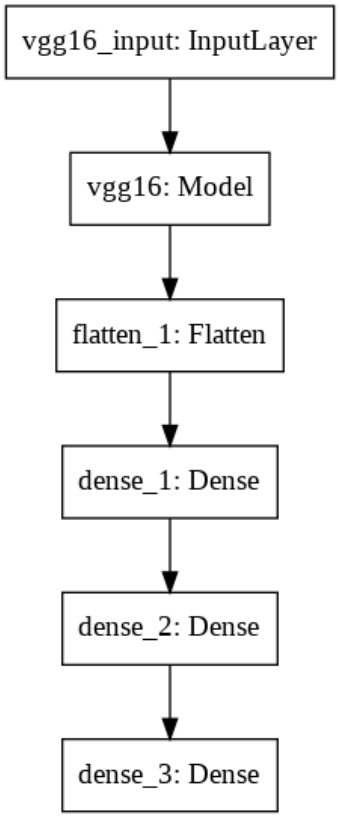
I used all conv layers from VGG and only replace the last classification layer (Dense layer)

```
conv_base.trainable = False
```

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 7, 7, 512)         14714688
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               12845568
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 27,691,841
Trainable params: 12,977,153
Non-trainable params: 14,714,688
```



Compile and fit the model using Python generator to augment the training data:

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])
```

```
#y_train = to_categorical(y_train, num_classes=2)       #No need to hot encode if you use binary
#y_valid = to_categorical(y_valid, num_classes=2)

# applying data augmentation on train set
datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

datagen.fit(x_train)
```

```
validation_datagen = ImageDataGenerator()       # no augmentation for validation set
validation_datagen.fit(x_valid)
```

Training:

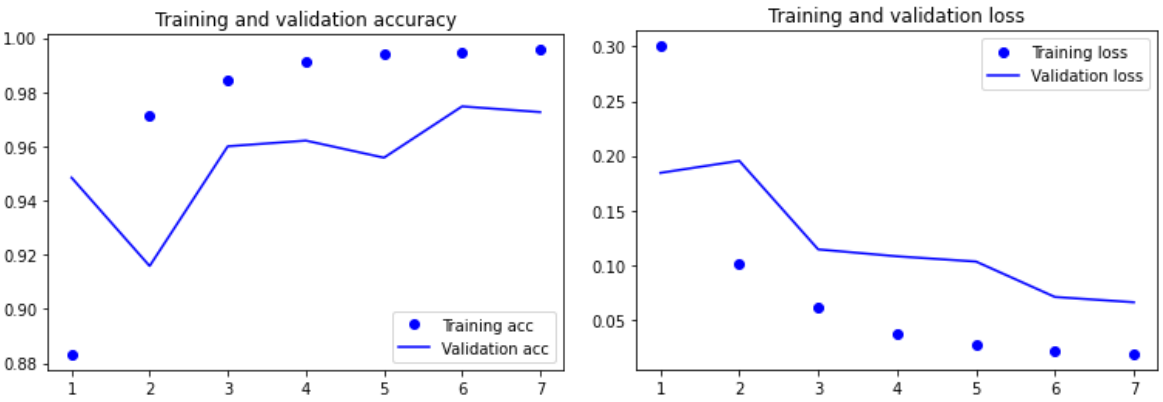
```
[17] epochs = 7   # epoch 6 , batch_size =32
     history = model.fit_generator(datagen.flow(x_train, y_train, batch_size=50),
                         steps_per_epoch=len(x_train) / 32, epochs=epochs,
                                  validation_data=validation_datagen.flow(x_valid, y_valid, batch_size=32))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:1033: The name tf.assign_add is de

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:1020: The name tf.assign is depre

Epoch 1/7
70/69 [==============================] - 49s 704ms/step - loss: 0.3010 - acc: 0.8834 - val_loss: 0.1844 - val_acc: 0.9485
Epoch 2/7
70/69 [==============================] - 41s 591ms/step - loss: 0.1012 - acc: 0.9718 - val_loss: 0.1954 - val_acc: 0.9159
Epoch 3/7
70/69 [==============================] - 44s 626ms/step - loss: 0.0608 - acc: 0.9846 - val_loss: 0.1144 - val_acc: 0.9600
Epoch 4/7
70/69 [==============================] - 43s 619ms/step - loss: 0.0375 - acc: 0.9917 - val_loss: 0.1081 - val_acc: 0.9621
Epoch 5/7
70/69 [==============================] - 43s 620ms/step - loss: 0.0273 - acc: 0.9940 - val_loss: 0.1033 - val_acc: 0.9558
Epoch 6/7
70/69 [==============================] - 43s 616ms/step - loss: 0.0213 - acc: 0.9949 - val_loss: 0.0709 - val_acc: 0.9748
Epoch 7/7
70/69 [==============================] - 43s 618ms/step - loss: 0.0186 - acc: 0.9957 - val_loss: 0.0660 - val_acc: 0.9727
```

Plots of accuracy and Loss for training and validation:



The model weights and .json file are saved so they could be loaded later for evaluating test data:

To evaluate the model, the model weight and json file is loaded and the trained model is evaluated on test data.

```
scores = model.evaluate(x_test, y_test)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
200/200 [==============================] - 56s 280ms/step
accuracy: 40.00%
```

```
predictions = model.predict(x= x_test)
```

1. **Method 1: CNN + LSTM**

This method consider timeseries. So For this method I used TimeDistributed layer to wrap up every 10 frames in the LSTM layer. Since I had a fixed parameter in my TimeDistributed layer, I had to manually adjust the number of frames. (The total number of frames in this case should be coefficient of 10) Note that the same thing need to be consider while evaluating the model with test data.

```
from keras.models import Sequential
from keras.layers import Conv2D, Dropout,Flatten, MaxPooling2D, BatchNormalization, TimeDistributed
from keras.layers import LSTM, Dense


n_timesteps = 10
lstm_cells =5

model = Sequential()
model.add(TimeDistributed(Conv2D (16, (3,3), activation = 'relu'),
                                  input_shape = (n_timesteps, 250,250,3)))
model.add(TimeDistributed(Conv2D (32, (3,3), activation = 'relu')))
model.add(TimeDistributed(Dropout(0.5)))
model.add(TimeDistributed(MaxPooling2D((2,2))))
model.add(TimeDistributed(Dropout(0.5)))
model.add(TimeDistributed(BatchNormalization()))
model.add(TimeDistributed(MaxPooling2D((2,2))))
model.add(TimeDistributed(Flatten()))

model.add(LSTM(units = lstm_cells, return_sequences = True, dropout=0.1, recurrent_dropout=0.5))
model.add(Dense(1, activation='sigmoid'))
#model.summary()
```

The challenge of using TimeDistributed layer is the input shape of the dataset. So The x_train and y_train are reshaped for this method:

```
# Chaning the shape of the input for TimeDistributed layer in the model
x = []
y = []
for i in range(0, 240, 10):
    x.append(x_train[i:(i+10)])
    y.append(y_train[i:(i+10)])

x = np.asarray(x)
y = np.asarray(y)
print(x.shape)
print(y.shape)

x_train = x
y_train = y
```

```
(24, 10, 250, 250, 3)
(24, 10)
```

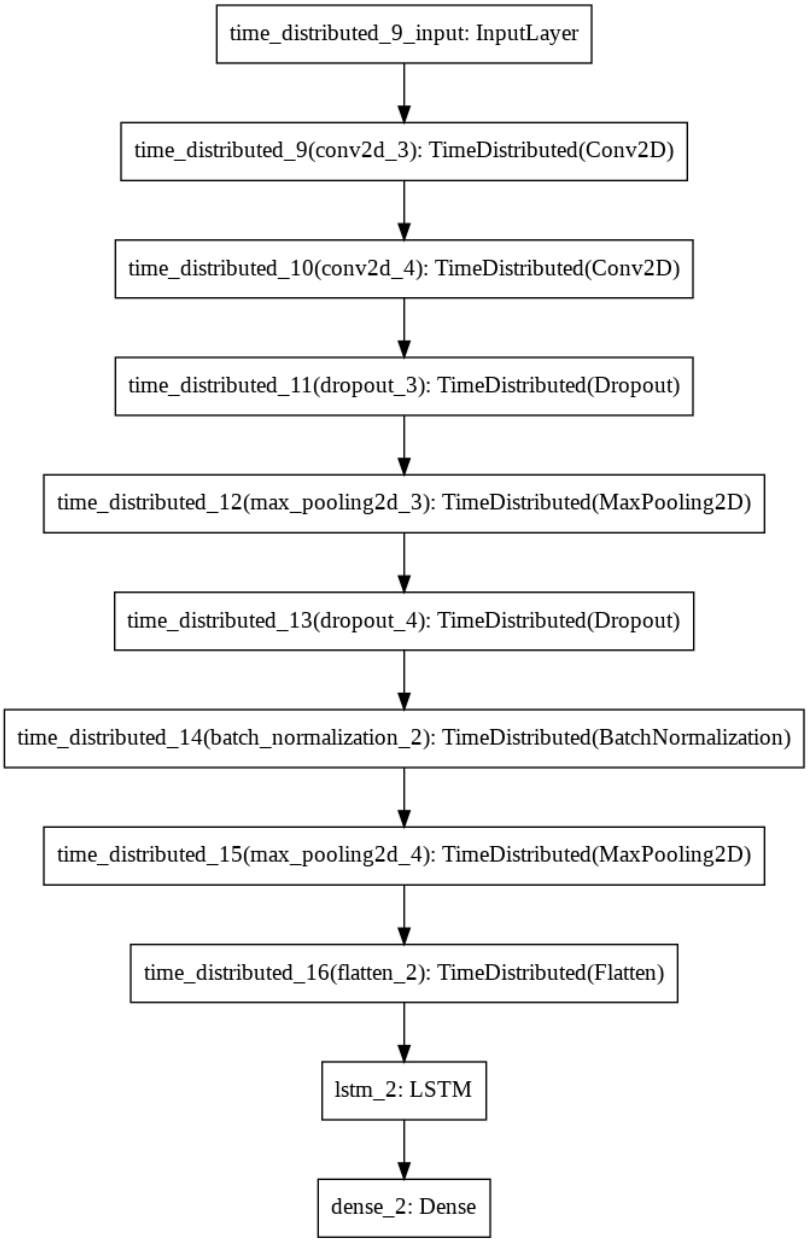
```
y_train = y_train.reshape(24, 10,1)
print(y_train.shape)
```

```
(24, 10, 1)
```

Model Layers and parameters:

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
time_distributed_162 (TimeDi (None, 10, 248, 248, 16)  448
_________________________________________________________________
time_distributed_163 (TimeDi (None, 10, 246, 246, 32)  4640
_________________________________________________________________
time_distributed_164 (TimeDi (None, 10, 246, 246, 32)  0
_________________________________________________________________
time_distributed_165 (TimeDi (None, 10, 123, 123, 32)  0
_________________________________________________________________
time_distributed_166 (TimeDi (None, 10, 123, 123, 32)  0
_________________________________________________________________
time_distributed_167 (TimeDi (None, 10, 123, 123, 32)  128
_________________________________________________________________
time_distributed_168 (TimeDi (None, 10, 61, 61, 32)    0
_________________________________________________________________
time_distributed_169 (TimeDi (None, 10, 119072)        0
_________________________________________________________________
lstm_17 (LSTM)               (None, 10, 5)             2381560
_________________________________________________________________
dense_28 (Dense)             (None, 10, 1)             6
=================================================================
Total params: 2,386,782
Trainable params: 2,386,718
Non-trainable params: 64
_________________________________________________________________
```

Model Architecture:



Running the model for 10 epochs:

```
[71] epochs = 10
     history = model.fit(x=x_train, y=y_train, validation_split = 0.2, batch_size=15, epochs=epochs, shuffle=False)
```

```
Train on 253 samples, validate on 64 samples
Epoch 1/10
253/253 [==============================] - 12s 48ms/step - loss: 0.6907 - acc: 0.6945 - val_loss: 0.4184 - val_acc: 0.9594
Epoch 2/10
253/253 [==============================] - 7s 27ms/step - loss: 0.4988 - acc: 0.7466 - val_loss: 0.5137 - val_acc: 0.8469
Epoch 3/10
253/253 [==============================] - 7s 27ms/step - loss: 0.4056 - acc: 0.8115 - val_loss: 0.4711 - val_acc: 0.9266
Epoch 4/10
253/253 [==============================] - 7s 27ms/step - loss: 0.3457 - acc: 0.8435 - val_loss: 0.4928 - val_acc: 0.9547
Epoch 5/10
253/253 [==============================] - 7s 27ms/step - loss: 0.2818 - acc: 0.8996 - val_loss: 0.5078 - val_acc: 0.9313
Epoch 6/10
253/253 [==============================] - 7s 27ms/step - loss: 0.2424 - acc: 0.9174 - val_loss: 0.5734 - val_acc: 0.8563
Epoch 7/10
253/253 [==============================] - 7s 27ms/step - loss: 0.2087 - acc: 0.9486 - val_loss: 0.5680 - val_acc: 0.8688
Epoch 8/10
253/253 [==============================] - 7s 27ms/step - loss: 0.1853 - acc: 0.9581 - val_loss: 0.5597 - val_acc: 0.8766
Epoch 9/10
253/253 [==============================] - 7s 27ms/step - loss: 0.1771 - acc: 0.9597 - val_loss: 0.5517 - val_acc: 0.9219
Epoch 10/10
253/253 [==============================] - 7s 27ms/step - loss: 0.1553 - acc: 0.9731 - val_loss: 0.5738 - val_acc: 0.8813
```

Plotting Results:

```
<matplotlib.legend.Legend at 0x7f7d80608b38>
```

Evaluating the model on test video:

```
scores = model.evaluate(x_test, y_test)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
20/20 [==============================] - 10s 490ms/step
accuracy: 59.00%
```

```
predictions = model.predict(x= x_test)
```

The LSTM model shows a better accuracy compare to the CNN model but still not good.

Another challenge with this model is that when I rant for more epochs (20 -30)

The accuracy of the validation decreased to very low accuracy. Also the behavior of the model was also very fluctuating each time that I ran the model. One reason is that the number of parameters are too many in this model and the number of data is very few for such deep learning model. One wat to improve the model is to use transfer learning in the CNN part.

The timelabel.jason file is only generated for test data. Since with the method that I used for training each video present either pushing or other creating the timelabel file is kind of nonsense.

To test the model:

1. Load model weight (h5) and Jason file
2. Compile the model
3. Get x_test and test label (get frames from video, get numpy array and reshape them to the correct size)
4. Finally evaluate the test data