

Project Report

Topic:

The project assigned to me is **Heart Attack Detection**. This project presents a proposal to identify people with an apparent heart attack by detecting characteristic postures of heart attack. The method of identifying infarcts (in this case, possible heart attack) makes use of convolutional neural networks. I have built a CNN model and trained it on a GPU-enabled server (Google Colab) to recognize heart-attack from a video. I have trained the model with a specially prepared set of images from the internet and self-made that contain people simulating a heart attack. The promising results in the classification of infarcts show 91.75% accuracy and 92.85% sensitivity.

Dataset:

I created my own dataset by taking images of myself and my friends and also many from the internet. All the pictures contain only one person. In images labelled as an infarct, all images show a person's posture in which they have one or both hands on the chest. As regards to no infarct situations, images where people are performing daily activities were used. The initial image data set consisted of a total of 1520 images, 760 images of class "Infarct" and 760 images of class "No Infarct". For better learning and extracting of features, the number of images in the data set is increased using data augmentation. The following steps were performed:

- Each image was scaled to a maximum size of 256×256 pixels, maintaining the original proportion, both for "Infarct" and "No Infarct" images. This is in order to reduce the amount of data to be processed during the augmented data technique.
- After that, the images were classified into two categories, "Infarct" and "No Infarct". Furthermore, each category was split into the three subcategories of training, validation, and testing, as shown in Table 1.
- As the CNNs only have to infer a possible heart attack, people were extracted from the background of the image by reducing the noise caused by the variation of the background in order to improve the training set.
- People were automatically located in each image. For this purpose, I performed the instance segmentation and background removal and replaced the pixels with magenta color so as to be a contrast to the person. This is shown in Figure 1.
- Data augmentation is a process for generating new samples by transforming training data. In this case, each original picture generated 20 more different images. For this, six transformations were combined and applied to each image

(rotation, increase/decrease in width or height, zoom, horizontal flip, and brightness change). This is shown in Figure 2.

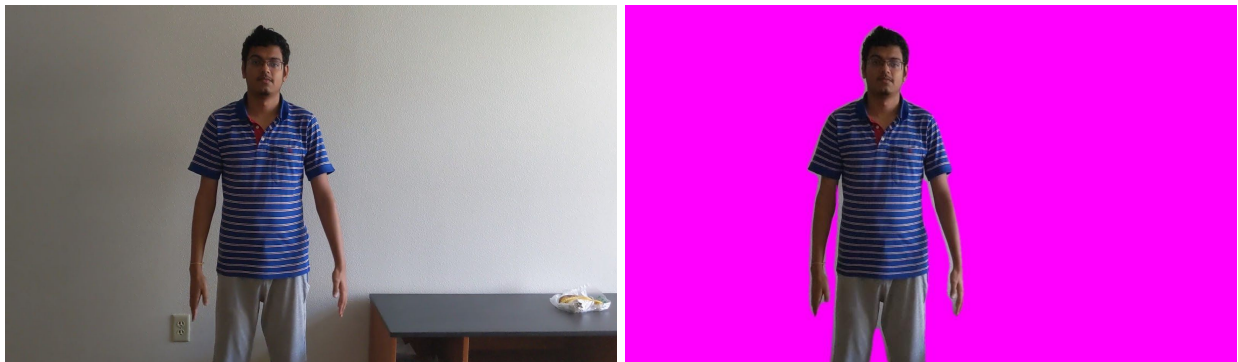


Figure 1: Instance segmentation and background removal

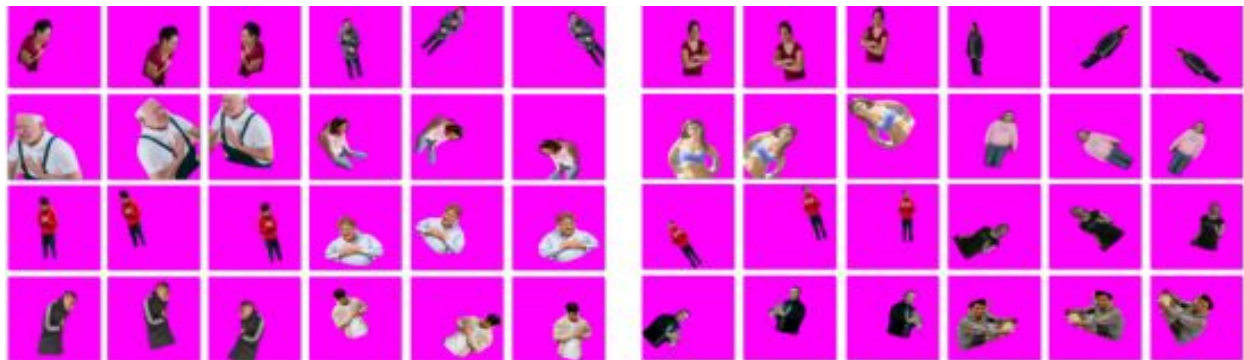


Figure 2: Data Augmentation

CNN Model:

Architecture:

At the beginning of the network, the architecture has five convolutional blocks, where each block is firstly composed of a convolution layer to highlight the general features in the image. Then, a max pooling layer is provided to keep the number of variables of the network low, in this way maintaining a size easy to compute.

In the middle of the network, just after the convolution blocks, there is a dropout layer that prevents the generated model from presenting an envelope training, mainly due to the limited amount of data. After this, a flatten layer allows changing the 2D design of the convolutional layers to a vectorial one so that the values generated in the previous layers are passed to the traditional neuron layers.

At the end of the network, ten layers composed of traditional neurons are arranged, each with 128 neurons, which deliver the result of forward propagation to a softmax function

with two outputs. These will classify whether there is or is not a person with a heart attack in the image.

A snippet of code of constructing the model/network is shown as follow:

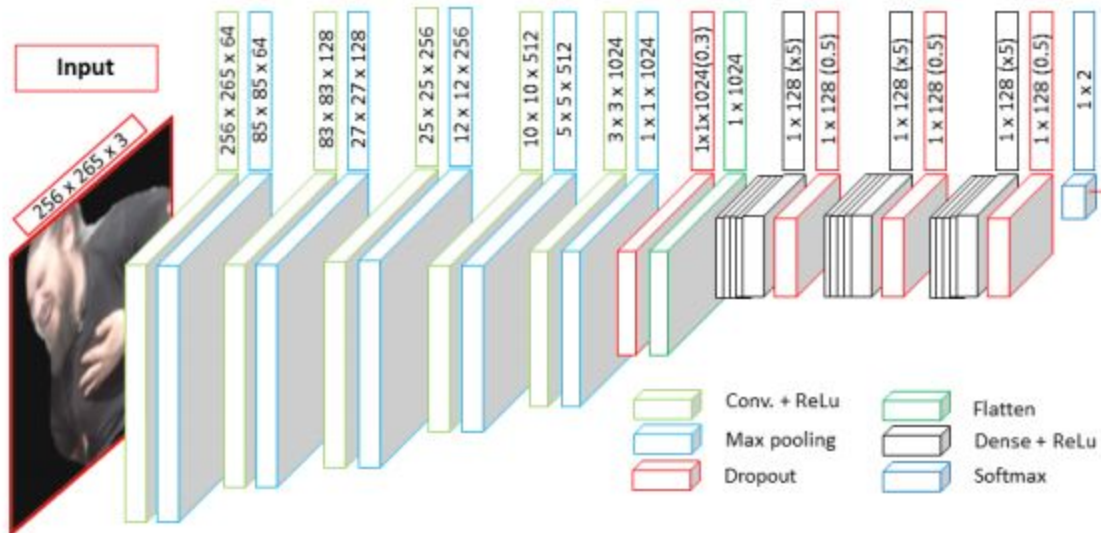


Figure 3: Neural Network Architecture

```
NET = Sequential()
NET.add(Convolution2D(64, kernel_size=(3, 3), padding = "same", input_shape=(256, 256, 3), activation='relu'))
NET.add(MaxPooling2D((3, 3), strides=(3, 3)))
NET.add(Convolution2D(128, kernel_size=(3, 3), activation='relu'))
NET.add(MaxPooling2D((3, 3), strides=(3, 3)))
NET.add(Convolution2D(256, kernel_size=(3, 3), activation='relu'))
NET.add(MaxPooling2D((2, 2), strides=(2, 2)))
NET.add(Convolution2D(512, kernel_size=(3, 3), activation='relu'))
NET.add(MaxPooling2D((2, 2), strides=(2, 2)))
NET.add(Convolution2D(1024, kernel_size=(3, 3), activation='relu'))
NET.add(MaxPooling2D((2, 2), strides=(2, 2)))
NET.add(Dropout(0.3))
NET.add(Flatten())

for _ in range(5):
    NET.add(Dense(128, activation='relu'))
NET.add(Dropout(0.5))

for _ in range(5):
    NET.add(Dense(128, activation='relu'))
NET.add(Dropout(0.5))

for _ in range(5):
    NET.add(Dense(128, activation='relu'))
NET.add(Dropout(0.5))

NET.add(Dense(CLASSES, activation='softmax'))

sgd = SGD(lr=LR, decay=1e-4, momentum=0.9, nesterov=True)

NET.compile(optimizer=sgd,
            loss='binary_crossentropy',
            metrics=['acc', 'mse'])
```

Input:

I chose the 70%–15%–15% for training, validation, and testing, which is a typical configuration in many other applications based on neural networks. The number of images and the distribution is given in Table 1. Once the training images were selected, the model was built using the Tensorflow framework. A precision of 99% was achieved during training, which allowed 91.75% accuracy and 92.85% sensitivity in the test set, defining a learning rate of 0.003 and using the gradient descent optimiser.

Class	Initial	Training	Initial	Validation	Initial	Testing	Final
	Training	Augmented	Validation	Augmented	Testing	Augmented	
Infarct	532	10,640	114	2280	114	2280	15,960
No Infarct	532	10,640	114	2280	114	2280	15,960
Total	1064	21,280	228	4,560	228	4560	31,920

Table 1 : Data distribution and total images after augmentation

The output for each hidden layers and the final output shape is described in Figure 4.

Hyperparameters:

In this project, there are four hyperparameters: batch size; epochs, dropout and learning rate.

Batch Sizes tried are 32, 48 and 64. The best result comes from Batch size 48 for now.

Epochs - 10,

Steps per epoch in training - 500,

Steps per epoch for validation - 100,

Dropout - 0.5

Learning Rate - 0.3.

As of now, these set of hyperparameters give the best accuracy. More fine-tuning can be tried on the hyperparameters to try and improve the model more but it takes time to train the full model everytime on Google Colab. So, experimentation with hyperparameters is still a work in progress but the existing results are very promising.

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 256, 256, 64)	1792
max_pooling2d_1 (MaxPooling2D)	(None, 85, 85, 64)	0
conv2d_2 (Conv2D)	(None, 83, 83, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 27, 27, 128)	0
conv2d_3 (Conv2D)	(None, 25, 25, 256)	295168
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 256)	0
conv2d_4 (Conv2D)	(None, 10, 10, 512)	1180160
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 512)	0
conv2d_5 (Conv2D)	(None, 3, 3, 1024)	4719616
max_pooling2d_5 (MaxPooling2D)	(None, 1, 1, 1024)	0
dropout_1 (Dropout)	(None, 1, 1, 1024)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 128)	131200
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 128)	16512
dense_5 (Dense)	(None, 128)	16512
dropout_2 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 128)	16512
dense_7 (Dense)	(None, 128)	16512
dense_8 (Dense)	(None, 128)	16512
dense_9 (Dense)	(None, 128)	16512
dense_10 (Dense)	(None, 128)	16512
dropout_3 (Dropout)	(None, 128)	0
dense_11 (Dense)	(None, 128)	16512
dense_12 (Dense)	(None, 128)	16512
dense_13 (Dense)	(None, 128)	16512
dense_14 (Dense)	(None, 128)	16512
dense_15 (Dense)	(None, 128)	16512
dropout_4 (Dropout)	(None, 128)	0
dense_16 (Dense)	(None, 2)	258
Total params: 6,633,218		
Trainable params: 6,633,218		
Non-trainable params: 0		

Figure 4: Network Summary

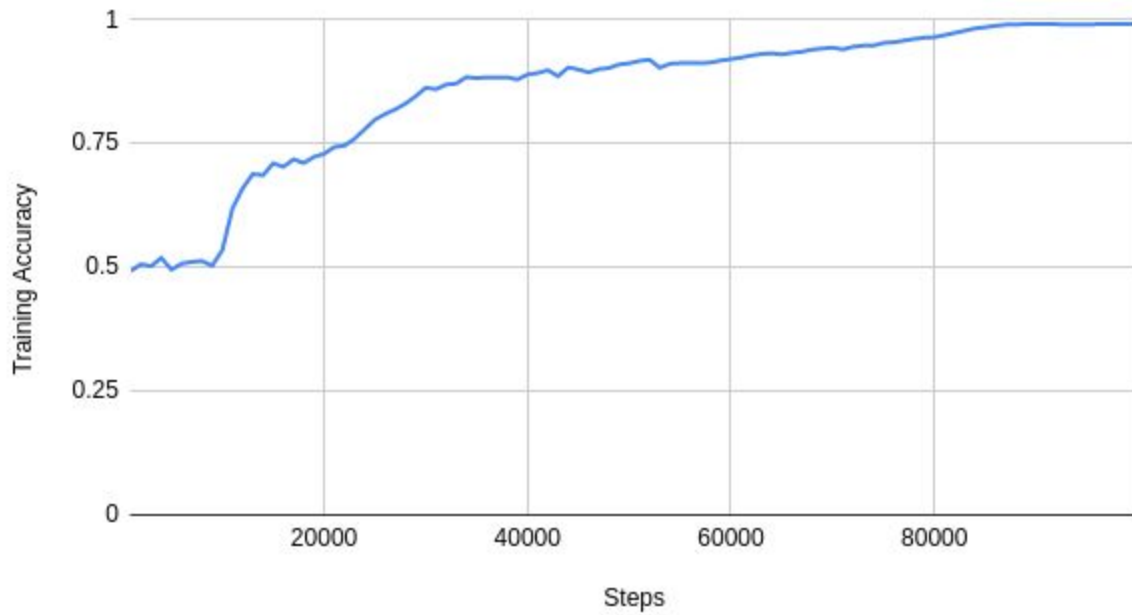


Figure 5: Training Accuracy vs. Steps

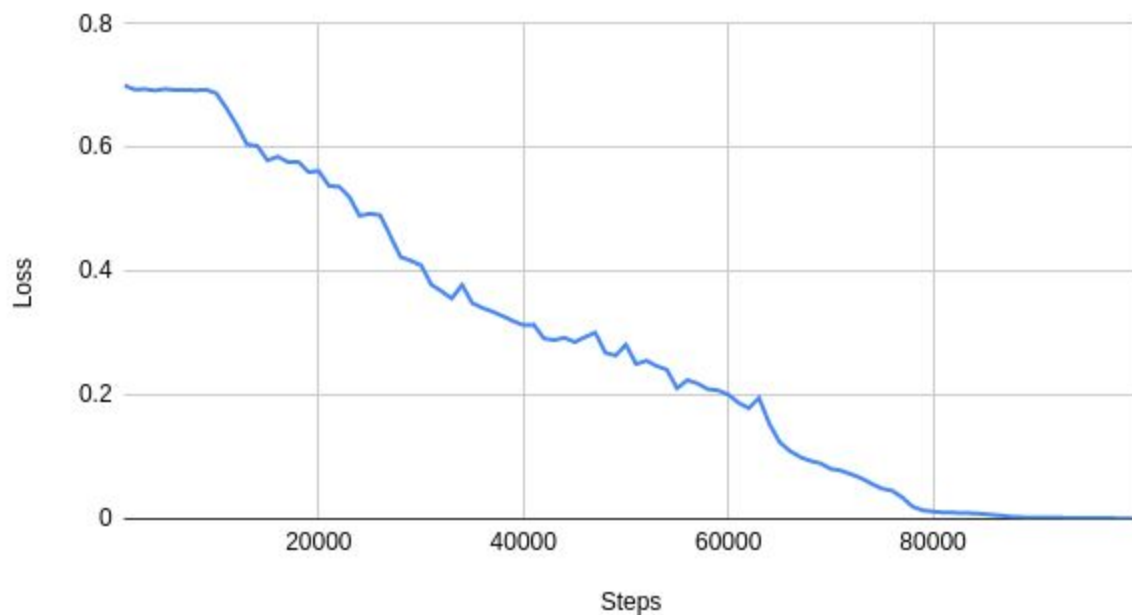


Figure 6: Training Loss vs. Steps

Drawbacks of the current model:

1. There are quite a number of false positives as seen in the testing of the videos. This is because when the hand or some part of the hand comes in front of the chest, as there is no 3D depth available, it sees the image as 2D and assumes

the hand is on the chest and detects it as positive. But when the side angle is seen, the network detects correctly. So, such images should be added to the training dataset has to be increased so that the network can learn to detect correctly.

2. In this current model, I haven't taken into account the facial expressions at this moment. So, it will classify a video of a person laughing with hands on or near the chest as a positive case of a heart attack. This is also a case of false positive. Work is needed in detecting facial expressions and combining the testing of that model with this existing model.
3. In this model, I'm detecting the presence of infarct in each frame of the video. The sample videos I have taken for testing are around 20-25 seconds long. The model takes around 10-15 minutes in total for the whole video. But if we pass just the frames as arguments to the python script, it will take around 15-20 seconds for each frame which is a decent enough time for real-time application.

Improvements from the current model:

1. Taking the video filename as an argument and automating the extracting frames, instance segmentation and background removal processes.
2. Increase the dataset size by adding images with hands just randomly in front of the chest classified as non-infarct and train the model again. The accuracy on the test dataset increased by 0.05%. But no significant difference was found in the plots and json files for videos.

Instruction on how to test the trained CNN:

The whole workflow i.e. codes, videos used, generated plots and json files can be found at the Github repo. <https://github.com/Kartikvenkat98/Heart-Attack-Detection>

For training the mode, you should run *train.py*. The model will be saved as *model.h5* in the */model* directory.

For directly testing the code on the test dataset using the pre-trained model, you can directly run *test.py*.

There are three steps in the whole process:

1. Getting frames from the video.
For this, you have to run *frame_generator.py path-to-video-file*.
This command will generate frames for a given video and save it in the */frames* directory.
For example, *python frame_generator.py ./videos/video_1.mp4*
2. Second step is the instance segmentation and background removal from each of the extracted frames. For this, we will run *seg_backrem.py*.
The background is changed to magenta color for having the contrast for the segmented image. This command will generate the segmented image for the frames and save them in */fg-extract* folder.
3. The final step is testing each of the background removed frames by running *test_modified.py path-to-video-file*.
This will also generate the required plot of the predicted probability of heart attack at a specific time instant and also creates the corresponding json file.
For example, *python test_modified.py ./videos/video_1.mp4*

Plan ahead for the next submission:

1. Continue tinkering with the hyperparameters to see if there is any more scope of improvement in the accuracy.
2. Keep adding more images in the training and validation dataset with hands near the chest and see if there is a decrease in the number of false positives.
3. Try to read more papers on RNN/LSTM in activity recognition to see if adding those layers can help increase the accuracy of the model or speed up the process of classification in videos.