Yoga Pose Image Classification

Final Report

Group 8: Jingjing Li, Jinglin Zhong, Shuyi Jiang, Wanxuan Zhang

1. Abstract

Image recognition has been one of the most important fields of image processing and computer vision, and human pose estimation would be the one encountered many challenges. Human activity analysis would be useful in many areas. Especially under the circumstance of the Covid-19 pandemic, human pose recognition would be helpful for people to do some self-instructed yoga and receive real-time feedback for daily simple exercise when staying at home all day long. In this case, our project would focus on the yoga poses, taking a look at the features of the yoga pose images, preprocessing the images for better training, building various deep learning models for pose classification and comparing results among different models. We have tried both building our own convolutional neural network as well as three pretrained models for transfer learning, including VGG16, ResNet50, and DenseNet201. By testing on the validation dataset to

compare different models' performance, we found the pretrained DenseNet model performs the best, with 1.33 loss and 64% accuracy on the validation dataset after training of 10 epoch. Finally, we visualized the classification result using Streamlit API and proposed to deploy and maintain our model on a weekly basis.

2. Business Problem

Human pose estimation is a challenging problem in computer vision because of the distraction from illumination variation, background clutterness, clothing variation and humans' interaction with surroundings. One use case of pose estimation is in exercise and fitness. With the continuation of COVID, humans begin to choose "smart home gym" instead of traditional membership gyms, which are equipped with high-tech capabilities made available with simply Wi-Fi and Bluetooth. The virtual fitness market still has a huge potential.

In one form of exercise, yoga, we all know the utmost importance to practice it in a correct posture as incorrect posture could be detrimental to the body. Therefore the deep learning model we propose on pose estimation is essential in yoga poses, which serves as the first step for equipment providers to give real-time personalized feedback to users and help them correct movements or suggest more suitable poses.

3. Data Loading and Initial Cleaning

This dataset needed lots of data cleaning and because of that, loading data was a recursive process.

We first deal with class labels by converting them from old classical Sanskrit language, which is the root of many Indian languages to English pose names for easy understanding.

Then we begin to process images. We first used the OpenCV package to read all the images in color mode but later found that some pictures are grayscale, some pictures are not images but GIFs,and some pictures are color images but with 4 channels (RGB and transparency). For the last case, if we read them in color mode, they would become complete black images when plotting. Therefore, in order to take into account all these

circumstances, we read in four categories separately: gray scale images, color images, color images with 4 channels and “images” that cannot be loaded with *OpenCV*. After this, we ended up with 5641 color images with 3 channels, 3 gray scale images (figure 1), 348 4-channels images and 2 GIFs.

Then we processed each of these categories separately. For GIFs, we converted them into PNG using PIL library. For gray scale images, we converted them into RGB colorspace. For normal color images, since OpenCV returns images with colorspace BGR, we have to convert them into RGB. For images with 4 channels, things were a little complicated. Some of them were “fake” 4-channel images and they could be converted to RGB colorspace easily using OpenCV library. For the other 5 4-channel images, if we simply converted their colorspace from BGRA to BGR or RGB, they would become all black. In order to get rid of their transparent background, we combined them with a pure white background and then converted their colorspace from RGBA to RGB(figure 2).
And Finally there are 5994 images with 107 yoga poses, which are the labels.

Figure 1

Figure 2

```
print(f"After preprocessing,there are total {len(color_labels)} images")
print(f"There are {len(pose_mapping.keys())} different poses in dataset")
```

```
After preprocessing,there are total 5994 images
There are 107 different poses in dataset
```

Figure 3

Although we did lots of data cleaning parts when we read in the data, they unfortunately cannot be used in modeling with Keras since these images all have different sizes. Loading such a diverse and complicated dataset into the right format that can be processed by Keras needs the low-level tensorflow API which we are not so familiar with.

4. Exploratory Data Analysis

The dataset takes up 1.2 GB. Counting the number of images in each of the classes, we can see most classes have number of images ranging from 30 to 70 (figure 4) and the range is from 18 to 90 images for all classes. Therefore there is some class imbalance problem as the model may not learn well from classes with fewer than 20 images.
The top 10 classes with most images can be seen from the right bar plot (figure 5):

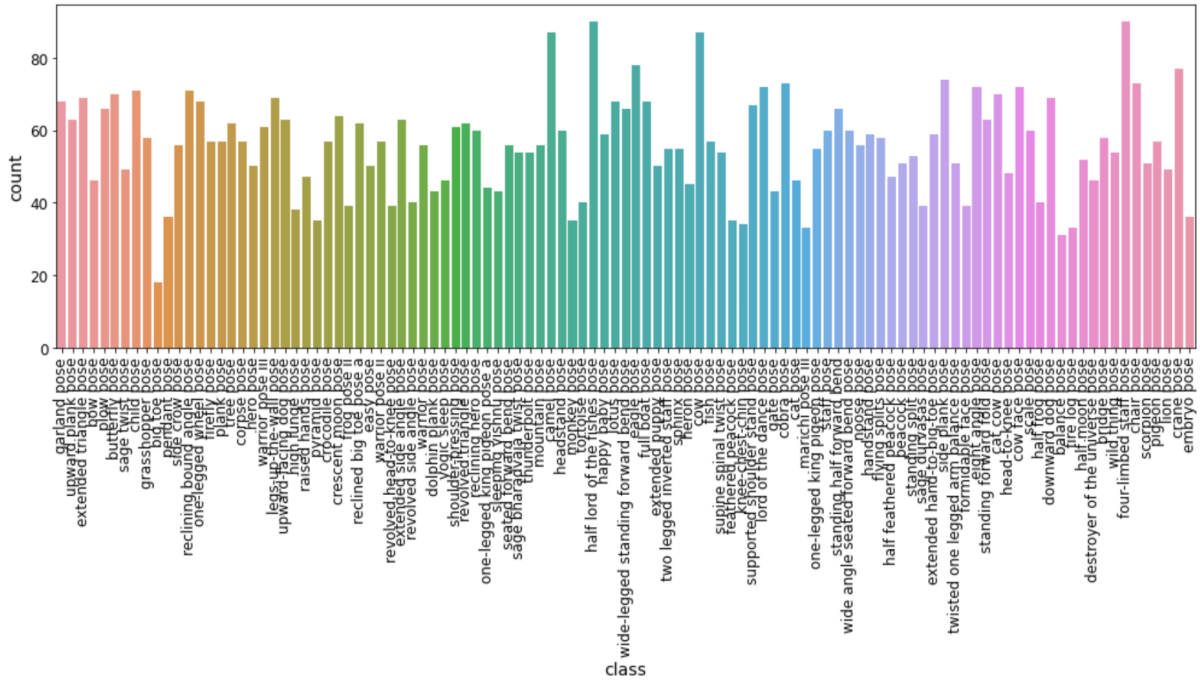


Figure 4

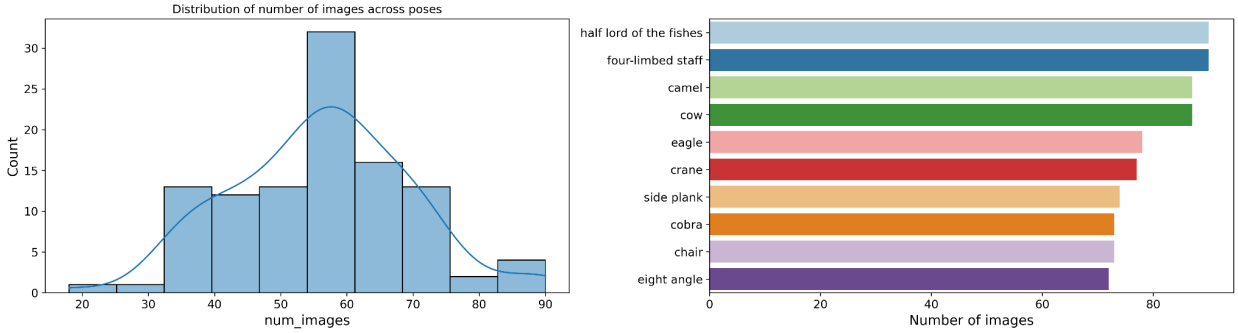

Distribution of number of images across poses
Count
num_images
half lord of the fishes
four-limbed staff
camel
cow
eagle
crane
side plank
cobra
chair
eight angle
Number of images

Figure 5

We plotted some random images from these 10 classes(figure 6).

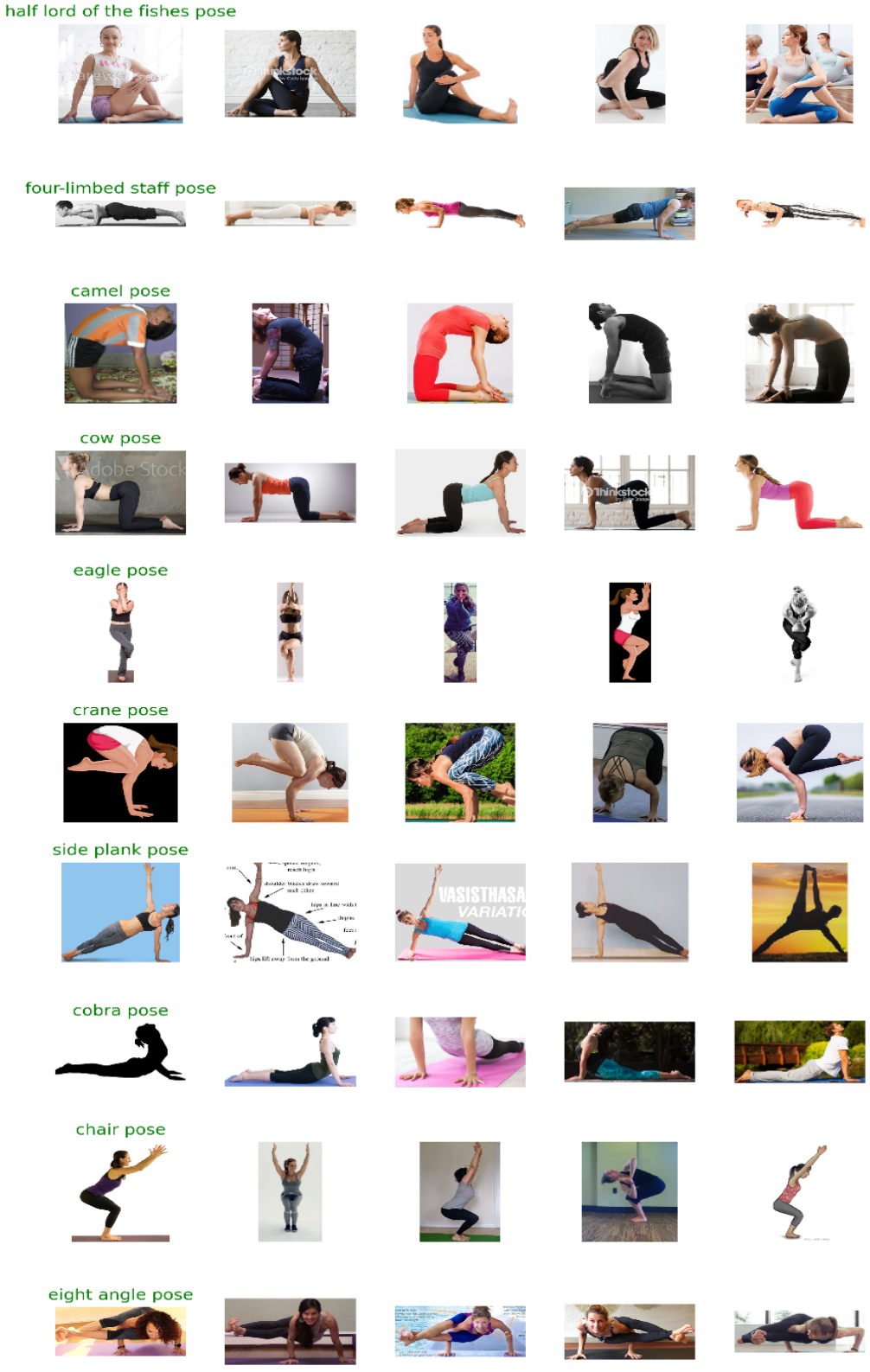


Figure 6

We can see images have different shapes, directions, colors and some of them even have watermarks. All of these elements could pose some difficulty to our final modeling.

Specifically, for the first image in class ‘garland pose’, we take a look at its type and shape, and the type of the image is a numpy N-dimensional array, with data type as float 32, and the height is 744, width is 496 and channels as 3 (figure 7, 8).

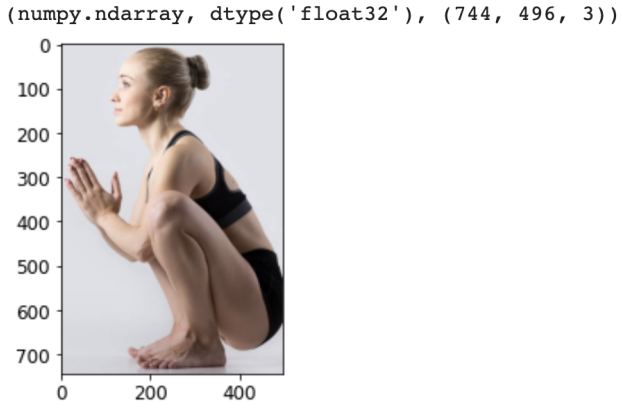


Figure7

Figure 8

We tried several image processing techniques, such as thresholding, blurring with different filters, and edge detection.

Since thresholding can only be done to gray scale images, we picked some images that have different illumination or have a very dark background. We applied global thresholding, Otsu’s thresholding and adaptive thresholding and compared their effects on different images. We found out Otsu’s thresholding always outperforms other methods: it enhances the images by making them clearer and brighter (figure 9, 10, 11), and even

remove annotations which are not overlapped with the main pose in the picture (figure 12).

Figure 9

Figure 10

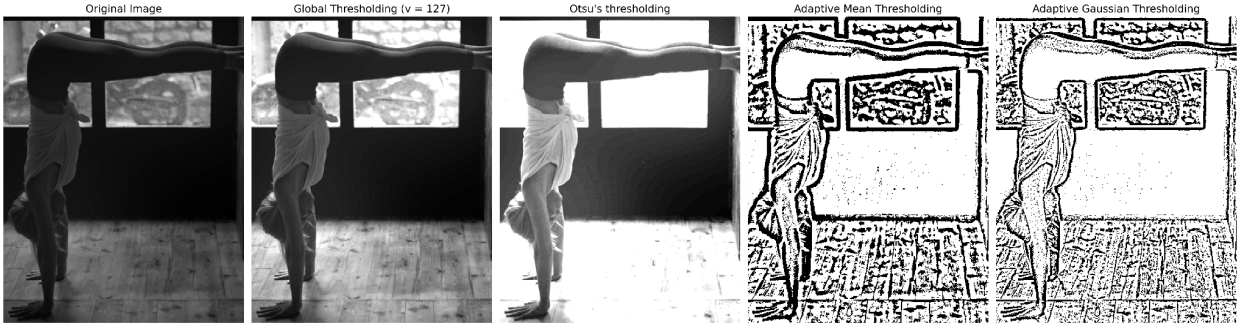


Figure 11

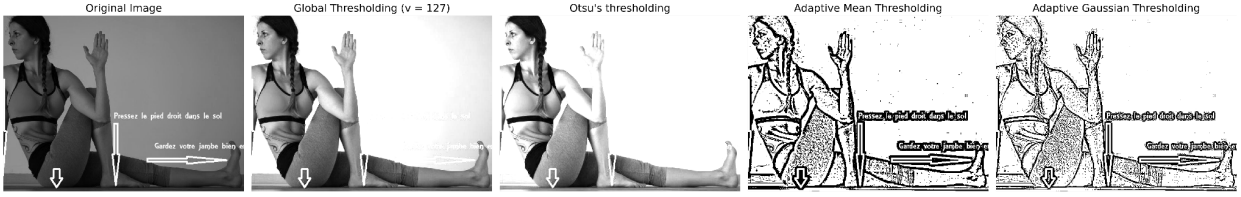


Figure 12

Regarding filtering, we tried averaging, Gaussian, median and bilateral filtering in order to remove noise in the images. Unfortunately, most images don't have the noise that can be alleviated by these filters, neither random points nor salt-and-pepper noise. Some common problems that these images have are varying color and illumination, which can

be reduced by thresholding. But regarding to crowded/noisy background, these filters do not help much (figure 13). But we can see that the gaussian kernel has the most blurring effect and the bilateral kernel indeed prevents the edges from blurring out.

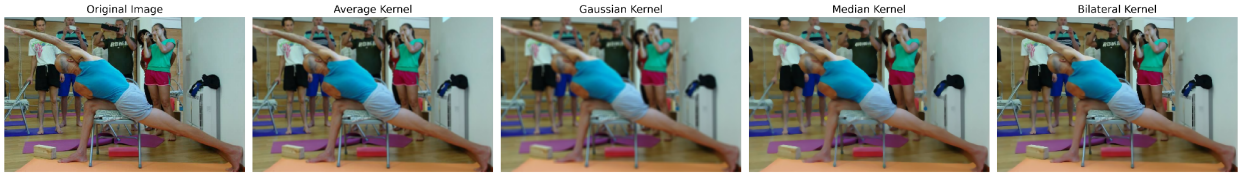
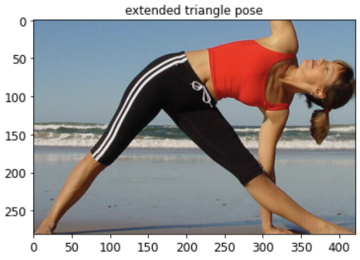


Figure 13

For color pictures in our dataset, we plotted a histogram for each color channel in a single plot.



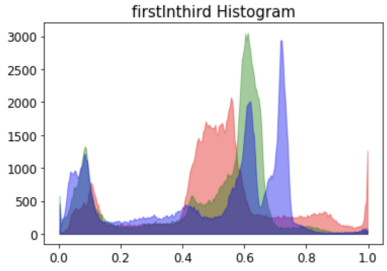


Figure 14

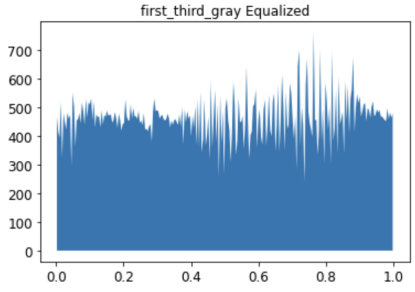


Figure 15

From the histogram (figure 14), we can see very sharp spikes here, and the pixel intensities are not uniformed. In this case, we would first try the histogram equalization which increases the global contrast of the image, especially when the image is represented by a narrow range of intensity values, and we would expect the adjusted histogram to have intensities better distributed throughout the plot. In our case, converting the inputting image to grayscale, and the output is the adjusted equalized histogram, which has few spikes left and is more uniformed (figure 15). The adjusted image turned out to have strong global contrast (figure 16). In most cases, histogram equalization might not be a good choice and would lose much information when input images have a large area low-intensity background.

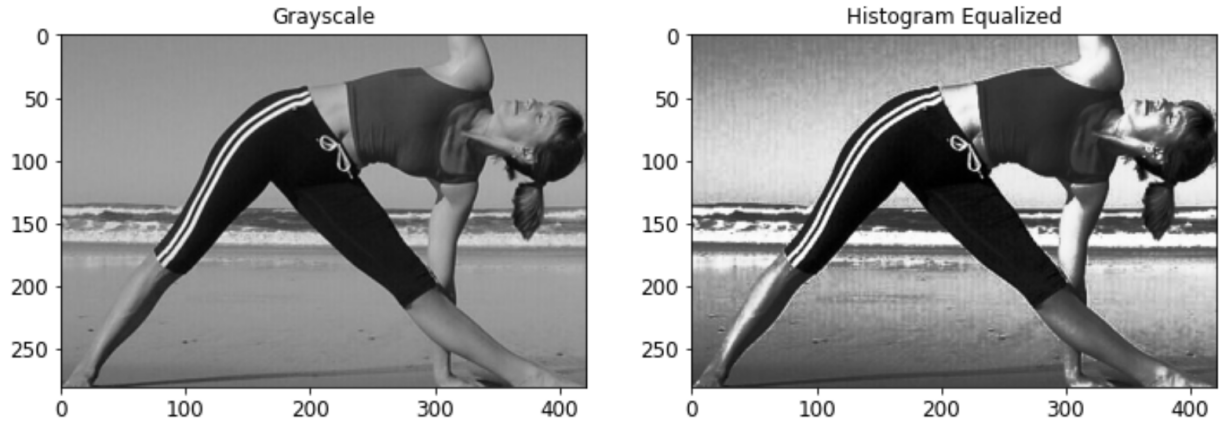


Figure 16

In this case, we would also try the contrast limited adaptive histogram equalization, which differs from conventional histogram equalization in that the adaptive method divides the image into small blocks like tiles and then histogram equalizes each of them. Also, to avoid overamplify in small tiles, contrast limited AHE would clip and distribute uniformly to other bins before doing histogram equalization when it detects any of them is above the specific contrast limit. This result (figure 17) should be better than the global histogram equalization and both the face and the background contrast has improved.

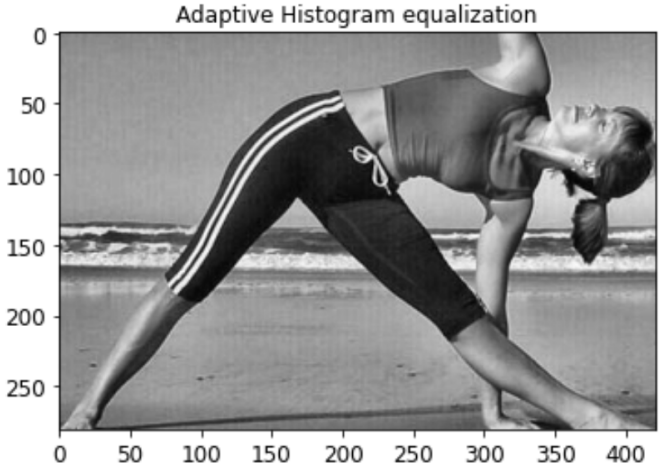


Figure 17

We also tried edge detection, which searches for borders between different colors and detects contours of objects in the images. In this case, the Sobel edge detection gives only the outlines of the object, and ignores other details in the images. From the plots (figure

18), we can see it helps to roughly outline the border of the person and the pose, but not very clear.

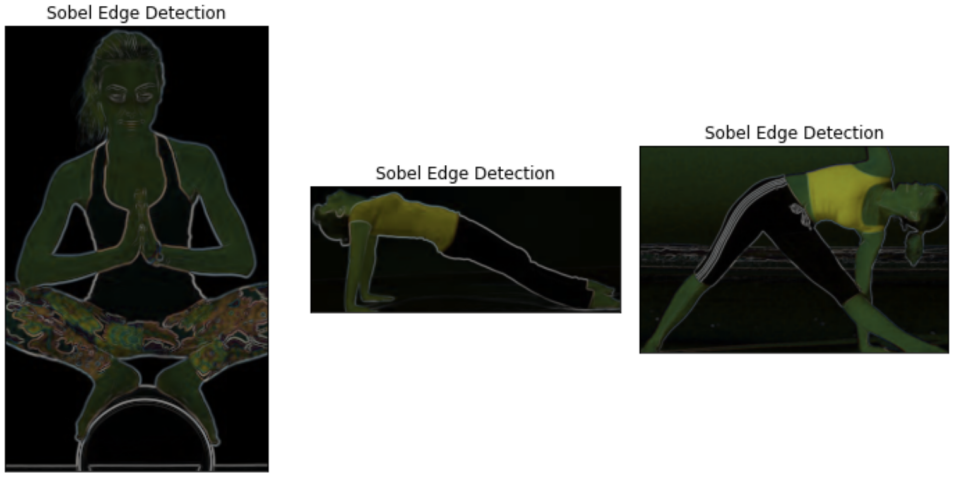


Figure 18

In addition, since the sizes of images in each class ranges from 18 to 90, number of images for some classes might not be large enough for modeling. In this case, we would also like to try the data augmentation to enrich our data amount, including flipping, rotating and zooming the images. Take a random image to see the effects (figure 19).

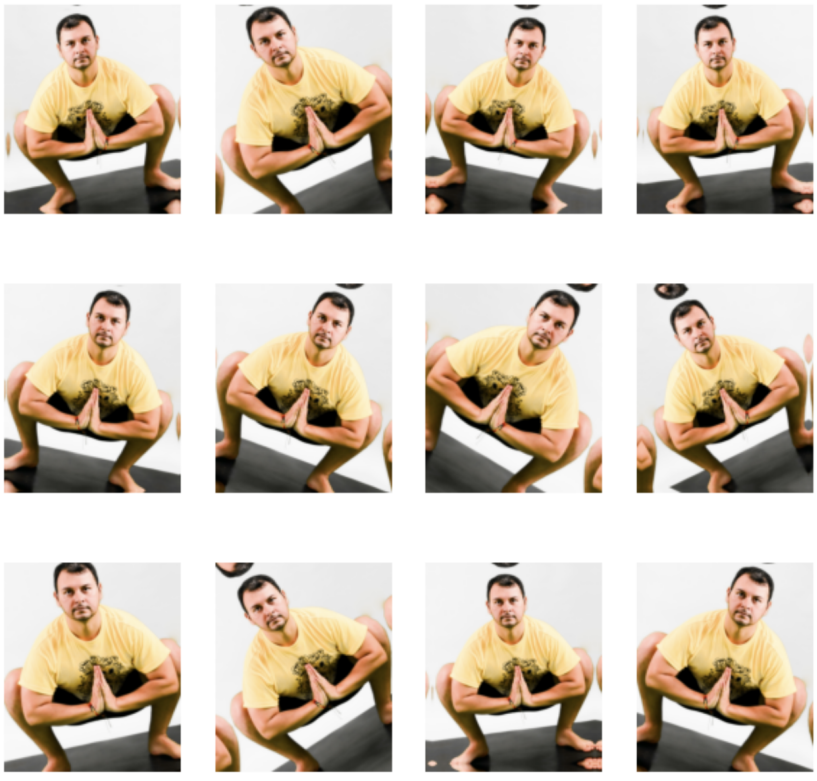


Figure 19

5. Model Building

Just as previously mentioned, since we could not directly apply our cleaned data into models, we decided to use Kereas ImageDataGenerator class to generate the dataset. This smart API could perform the real-time data augmentation while generating the Tensor images, so it lets us apply different transformations on the original image. Also the ImageDataGenerator class makes sure that the model would receive different versions of transformed image for every epoch. By doing this, our model could be more robust and better capture the main features.

```
train_datagen = ImageDataGenerator(rescale=1./255,
                                   width_shift_range = 0.1,
                                   height_shift_range = 0.1,
                                   shear_range = 0.1,
                                   zoom_range= 0.1,
                                   horizontal_flip=True,
                                   fill_mode='nearest',
                                   validation_split=0.15)
```

We would shift the image horizontally and vertically by 1% and set the shear intensity and range for random zoom to 1% as well. The generator would also flip the inputs horizontally randomly. After applying all the transformations, it would rescale the image and split the dataset into training and validation subset for the later measurements of model performance.

```
train_data = train_datagen.flow_from_directory(
    data_dir,
    target_size=(224, 224),
    color_mode='rgb',
    class_mode='categorical',
    batch_size=32,
    shuffle=True,
    seed=42,
    subset='training')
```

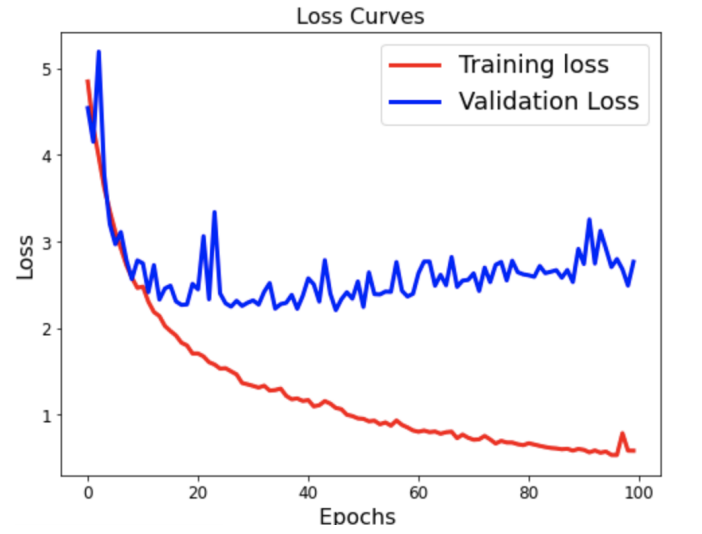
Then we make our generator connect to the target directory and ask it to resize every image to (224 * 224) and generates 32 batches. Finally, we got 5145 images for training and 848 images for validation.

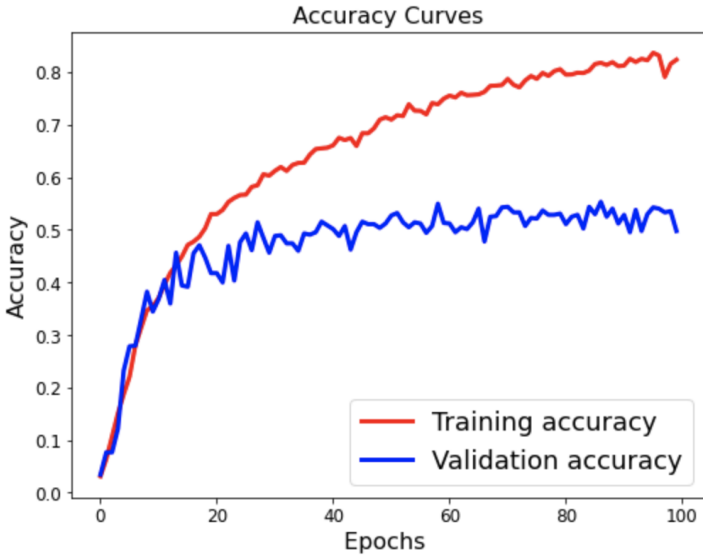
For the model part, we considered two ways to implement: building our own convolutional neural network and using pre-trained CNN models for transfer learning, and then we would compare these two methods using validation-accuracy and validation-loss to choose the best performance model.

1. CNN

```
model_cnn = tf.keras.models.Sequential([Conv2D(128,(3,3),input_shape=(224,224,3),activation='relu'),
                    BatchNormalization(),
                    Conv2D(128,(3,3),activation='relu'),
                    MaxPool2D(2,2),
                    Conv2D(64,(3,3),activation='relu'),
                    BatchNormalization(),
                    Conv2D(64,(3,3),activation='relu'),
                    MaxPool2D(2,2),
                    Dropout(0.25),
                    Conv2D(32,(3,3),activation='relu'),
                    BatchNormalization(),
                    Conv2D(32,(3,3),activation='relu'),
                    BatchNormalization(),
                    MaxPool2D(2,2),
                    Flatten(),
                    Dense(1024,activation='relu'),
                    Dense(512,activation='relu'),
                    Dropout(0.25),
                    Dense(107,activation='softmax')])
```

To build our own classification model, we used 2-dimensional convolution layers and relu as activation function. Then we added max-pooling layers and batch normalization after each convolution layer. We also added the dropout layer and set the dropout rate to 0.25 to prevent the overfitting. Since this is a multi-class project in which we need to predict 107 different yoga poses, we used the dense layer and softmax activation function as the output layer. To compile the model, we used nadam optimizer since usually it outperformed the others and used categorical_crossentropy as the loss function. After 100 epochs of training, the training accuracy achieved 83.65% while the validation accuracy only had approximately 54%.





From the graph, we could easily see that we still have a serious overfitting problem even though we have adopted a dropout layer. After about 20 epoches, the validation loss stopped decreasing. Our guess is that the model has learned too many noises or random fluctuations in the training data and we might handle this problem by reducing the network's capacity.

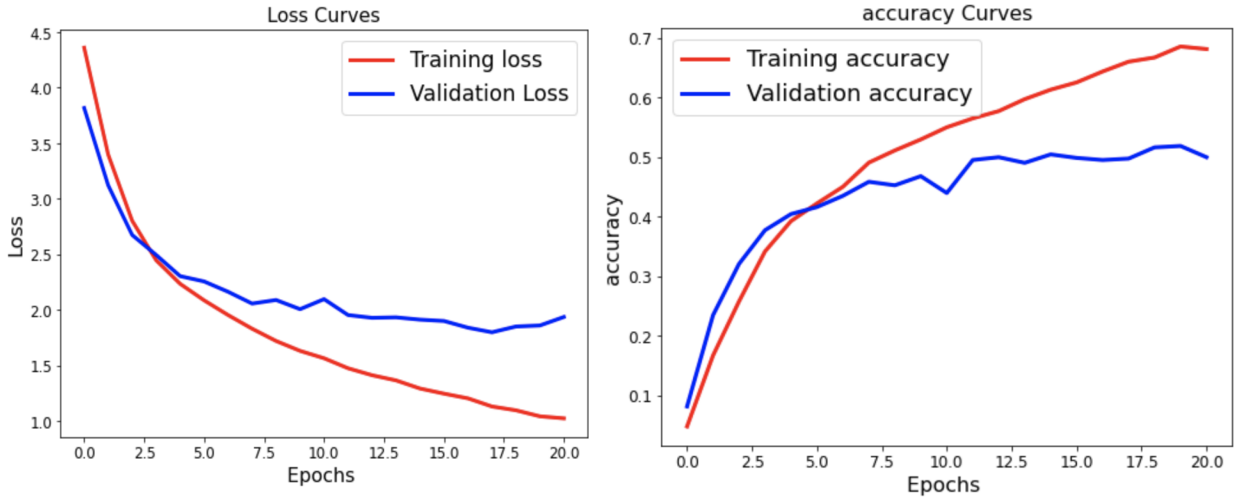
2. Transfer learning

We tried 3 pre-trained models: VGG16, ResNet50 and DenseNet201, and then built up our own customized layers. For all three models, we added a global average pooling layer, a flatten layer, dense layers with ReLU activation function and a dropout layer with dropout rate = 0.25 as we previously did.

```
base_model = keras.applications.DenseNet201(input_shape=(224, 224, 3),include_top=False, weights='imagenet')
x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(1024, activation="relu")(x)
x = Dropout(0.25)(x)
x = tf.keras.layers.Dense(512, activation="relu")(x)
outputs = tf.keras.layers.Dense(107, activation="softmax", name="classification")(x)
model_tf = tf.keras.Model(inputs=base_model.input, outputs = outputs)
```

This time we also added the callbacks API to use EarlyStopping class to stop training when our monitored metric val_loss stopped improving.

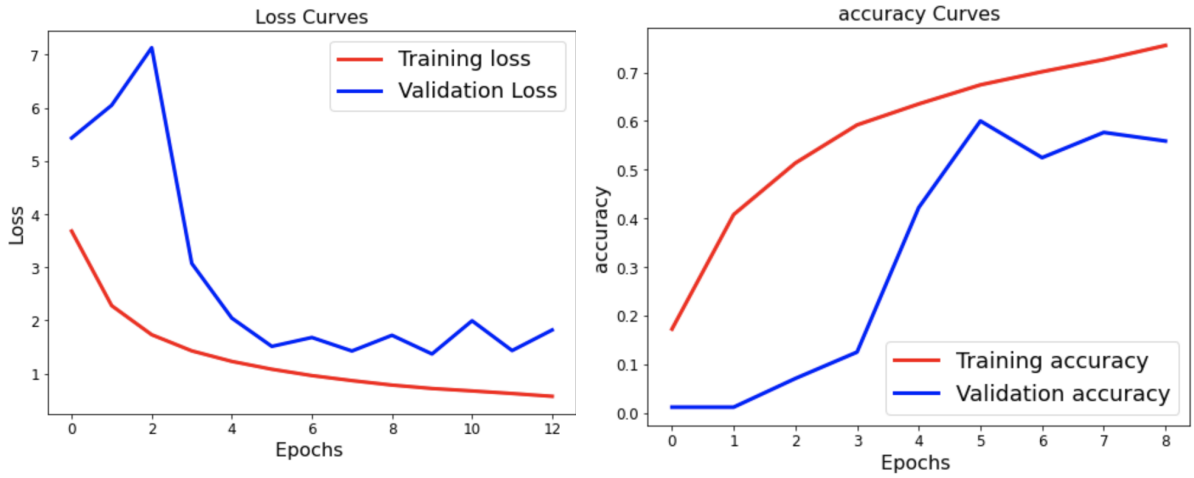
VGG16 is an advanced CNN model which has great understanding of image's shape, color and structure since it has been trained on millions of images with complex classification tasks. This is the main reason we considered this model. Also its structure is quite simple and we could choose how many layers to use based on our own needs. However, this model has 138 million parameters and is thus very heavy and needs a longer inference time. Compared with the other two models we chose, it indeed used more epochs (18 epochs) to achieve the best results.



The evaluation of the validation dataset has about 1.88 loss and 50.47% accuracy. Compared with our own model, its performance was not satisfactory and even had lower accuracy. Therefore, we considered applying the ResNet model.

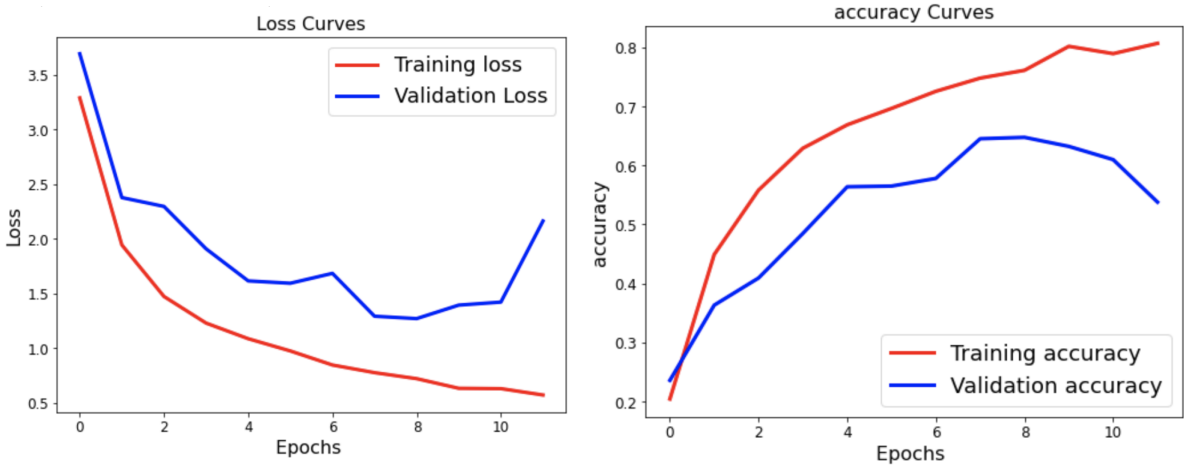
The ResNet model is also designed for image classification tasks and was proposed in 2015. This model tries to learn differences among learned features and it would zero out the weight if the learned features are not useful in the final decision and thus preventing overfitting. ResNet requires less memory than VGG and has faster inference time. In our

case, we applied ResNet50 which has 50 layers and it only used 6 epochs to reach the lowest val_loss.



The final model has 1.5223 loss and 0.5637 accuracy on the validation dataset.

Finally we considered the DenseNet model, in which each layer receives additional inputs from all previous layers and passes on its own feature maps to all later layers. What we value more here is that the DenseNet model could keep track of more diversified features and also maintains low complexity features. However, the dense concatenation is also requiring more GPU memory and needing more training time. In our model, it ran about 130s per epoch and longer than the other two models.



The final model of DenseNet had about 1.33 loss and reached 64% accuracy on the validation dataset. So far we have already found out our best performance model (see

table below). However it's worth noting that the val_accracy decreased a lot after the 10th epoch. The massive connections could harm the network's computation efficiency and parameter efficiency and make it more prone to overfitting.

Model	Training Time per Epoch	Validation Loss	Validation Accuracy
CNN Model	112s	2.7054	0.5236
VGG16	105s	1.8790	0.5047
ResNet50	116s	1.5223	0.5637
DenseNet201	130s	**1.3185**	**0.6403**

6. Model Management

To visualize the classification result, we used Streamlit, a user-friendly API, to interact with the audiences. We saved the DenseNet201 model architecture and parameters in a h5 file. After the user uploads a yoga pose image, the API will automatically preprocess the image, input into the loaded model and display the predicted class.

Regarding the model maintenance, we plan to deploy the model on the Google Cloud Platform. Our model will be updated once a week, by using the latest yoga pose images from active users to retrain the model and update the model parameters. Also, we will embed the model API into mobile APPs to help users receive real-time personalized feedback.

7. Conclusion

In this project, we focused on the yoga pose image classification task, preprocessed the images, applied data augmentation, built various deep learning models and compared their results. We implemented our self-defined convolutional neural network as well as other 3 pretrained models for transfer learning, including VGG16, ResNet50, and DenseNet201. We found the pretrained DenseNet201 model performs the best, with 1.33 loss and 64% accuracy. We think that it works best because reusing feature makes the network highly parameter-efficient and can keep track of more diversified features. But it is worth to mention that the dense concatenation needs more training time per epoch. In addition, too many epochs will decrease the accuracy and lead to overfitting, because of the worsen computation efficiency.

Finally, we visualized the classification result using Streamlit and proposed to deploy and update our best model in a weekly basis. This model API is essential in helping users practice yoga exercise in correct posture.

8. Future Works

Instead of using original pose images, we could do more preprocessing to get rid of background noise, such as only using contours or edges of poses, or extracting body keypoints/landmarks (ears, nose, wrist, hip, etc) using MoveNet model. Also we could first try to put similar yoga poses together, either through unsupervised clustering methods or human-label in order to reduce the number of target classes and increase accuracy. Furthermore, we could extend our single pose classification to multi-poses classification if multiple poses are presented in one image.

9. References

https://www.kaggle.com/code/remonboshra/yoga-posts-classification-model/data
https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035
https://cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/
https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html
https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_filtering/py_filtering.html