Project Report

Project Overview

The aim of this project is to build a model to classify yoga poses. To tackle the problem a supervised learning, single-label, image classification convolutional neural network algorithm is selected and trained on an image dataset containing 107 classes of yoga poses. The model endpoint is then connected to a web application to allow the user to input an image of a yoga pose and get its classification as response.

Image Classification is a fundamental task of Computer Vision (CV). It applies to a broad range of practical fields from autonomous driving [12] to fitness applications [11].

Neural network image classification algorithms are trained to classify the image pixels into one or more labels, i.e., to recognize the shape and pattern within a raw image [13]. However, such algorithms, as the meaning of their name Image Classification, can only "classify" images. For problems that aim on tracking real-time body movement there are the Pose Detection algorithms, e.g., PoseNet, DeepPose, and OpenPose. These algorithms identify the pose or movement by detecting the relative position of key points (coordinates of joints and key points) within the human body. Pose Detection is also a task of Computer Vision, and real-life applications of this method includes gaming, augmented reality, and workout monitoring applications [2, 18, 27, 28].

Although Pose Detection is not the focus of this work, a possible further improvement of this project could consider the linkage (coupling) of the image classification model studied here to a pose estimator algorithm, like PoseNet, as done in [18] for instance.

Problem Statement

The sequence of steps to develop this project is illustrated in Figure 1 below.

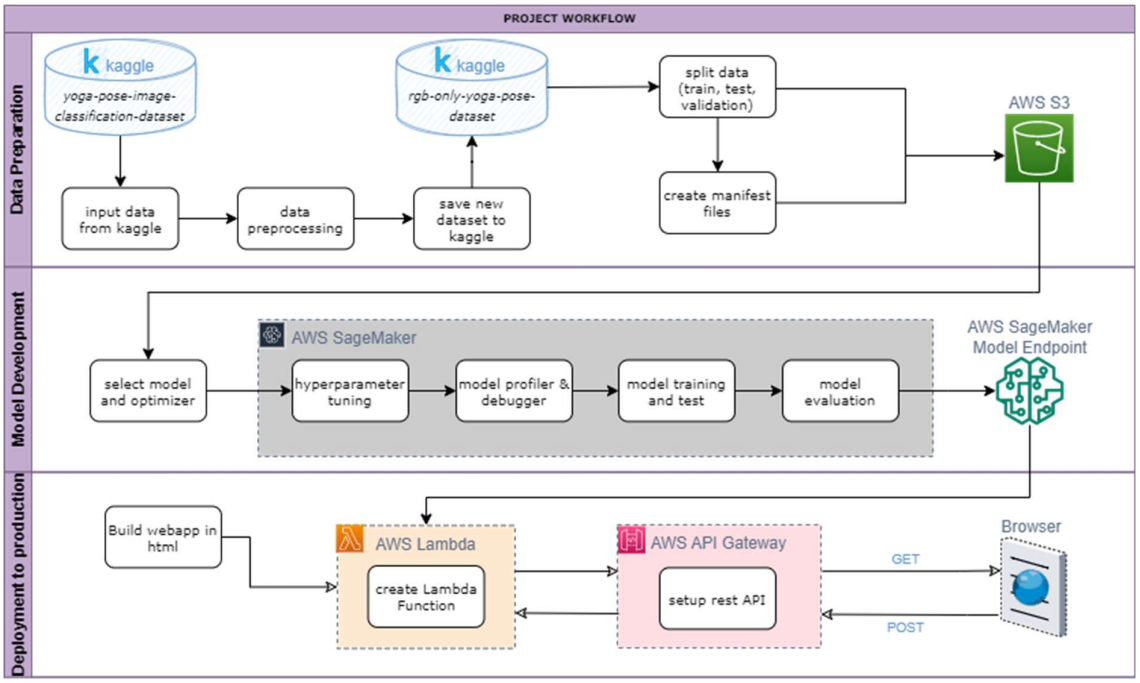

Figure 1: Project workflow

The project workflow is described below together with the algorithms, the techniques and AWS services utilized to develop this project.

Data Preparation phase:

During data preparation the dataset is downloaded and explored. Issues with image file extension and image file format are identified and fixed and the images mode are converted to 3-channel RGB mode. This preprocessed dataset is then uploaded to kaggle to be used in the next phase of the project. For more information on data preprocessing see the "Data Preprocessing" section bellow.

Next the 3-channel RGB mode images data is split into train, validation, and test sets (with the respective proportions of 0.7, 0.1, and 0.2) and manifest files are created. The data so structured is then uploaded to AWS S3 to be used during model training and validation.

Model Development phase:

The first part of this phase is the model selection, which is described in detail in the “Refinement” section below. After selection the model is then trained, finetuned, debugged and deployed in AWS SageMaker using boto3, Sagemaker SDK, and PyTorch ML framework. More info on SageMaker SDK, boto3, and PyTorch ML framework can be found in AWS documentation [3, 14, 22, 23].

Once the data is ready, the next step should be to train the model, however, not before finding a good model and setting-up its hyperparameters. The model used here was the pre-trained VGG16 with Adadelta gradient descent optimization (see "Algorithms and Techniques" section bellow for more information on model selection).

Finetuning is performed in a sequence of four Hyperparameter Tuning jobs with 6 training jobs each one. The first Hyperparameter Tuning job was followed by a Warm Start Hyperparameter Tuning job, which was performed to enlarge the number of hyperparameters tuned. The insights of these two previous tuning jobs allowed to narrow the hyperparameters range and therefore a second Hyperparameter Tuning job was launched followed by a second Warm Start Hyperparameter Tuning job (see section "Refinement" bellow for more information on hyperparameter tuning).

The best training job from hyperparameter tuning step was selected as the model for this project. This best model was submitted to debugging using Amazon SageMaker Debugger API. This step is useful to identify possible training issues. The SageMaker Debugger profiling report can be accessed at the link profiler-report (fsoaresantos.github.io).

The trained model was validated during model evaluation step using the images from validation set. See "Model Evaluation and Validation" section bellow for more info on this step.

Deployment to production phase:

A webapp is built to deploy the model into production. In this phase the user can finally use the trained model to classify an image of a yoga pose.

The webapp code is written in html, css, and javascript. Such code is placed within the lambda function package. The lambda function returns the html content as response to a GET request coming from API Gateway. In addition, the lambda function must invoke the SageMaker inference endpoint via a runtime object created using the boto3 library. The

lambda function returns the model prediction as a JSON content in response to a POST request.

In summary, the application is hosted via AWS Lambda and AWS API Gateway. To do so a rest API is setup in API Gateway with GET and POST methods configured to Lambda Function Integration type and Lambda Proxy Integration [4, 7, 24] (for more information about webapp see *yoga-pose-api.md* file in fsoaresantos/Yoga-pose-Image-Classification-Project (github.com)).

Figure below shows a flowchart of the application:

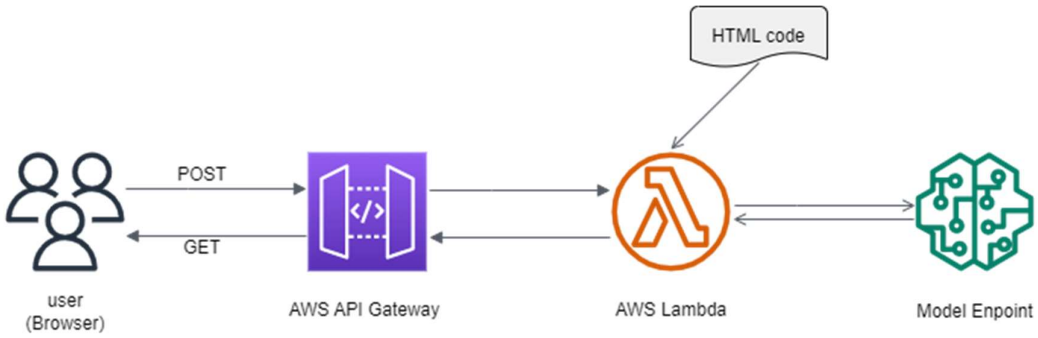

Figure 2: webapp flowchart diagram

Metrics

The key metric used to determine model performance was the average test loss. From model selection, to hyperparameter tuning, and model training and validation.

During model selection, the first step of model development, the test accuracy and average test loss of every model architecture experiment were plotted to contrast performance (see "Refinement" section below).

The average test loss was the metric used as final objective value during hyperparameter tuning to identify the best set of hyperparameters; and during model validation to compare benchmark model with this project model.

During model training information about average training for each epoch and average test loss and the test accuracy were observed. This project model displayed the following metrics values after 55 epochs were as follow:

```
Train Epoch: 55
Average training Loss: 0.058
Average test Loss: 0.129
Test accuracy: 35.162%
```

Data Exploration

The dataset used for developing this project comes from kaggle online platform and can be downloaded here [29]. It contains a total of 5994 sample images covering a range of 107 classes of images each one corresponding to a yoga pose.

The distribution of data between the 107 classes was observed. Nevertheless, no step was taken to tackle the problem of outliers and low represented classes in this project.

Here the data, i.e., the image files were checked for file extension, file format, and images mode with the aim of identifying inconsistences in these features [6, 15, 16]. The result found 5

different file extensions, 3 different file formats, and 5 different image modes as shown in the code output below.

```
5 file extensions found:
['png', 'jpeg', 'jpg', 'jpe', 'gif']

3 file formats found:
['PNG', 'JPEG', 'GIF']

5 image modes found:
['RGB', 'RGBA', 'P', 'L', 'LA']
{'RGB': 5627, 'RGBA': 341, 'P': 19, 'L': 3, 'LA': 4}
```

Exploratory Visualization

Exploratory data visualization focused on the visualization of a sample of images together with their file type, file extension and image mode and the visualization of the distribution of the number of images between the 107 classes of the dataset. The data presented issues on image file extension and a significant variation in image mode (Figure 3). The distribution of the number of images between classes was also variable (Figure 4 and 5).

Figure 3: Figure shows a sample of images from kaggle "yoga-pose-image-classification-dataset" with respective file extension and image mode. Note that the last RGB mode image has file extension .jpe.

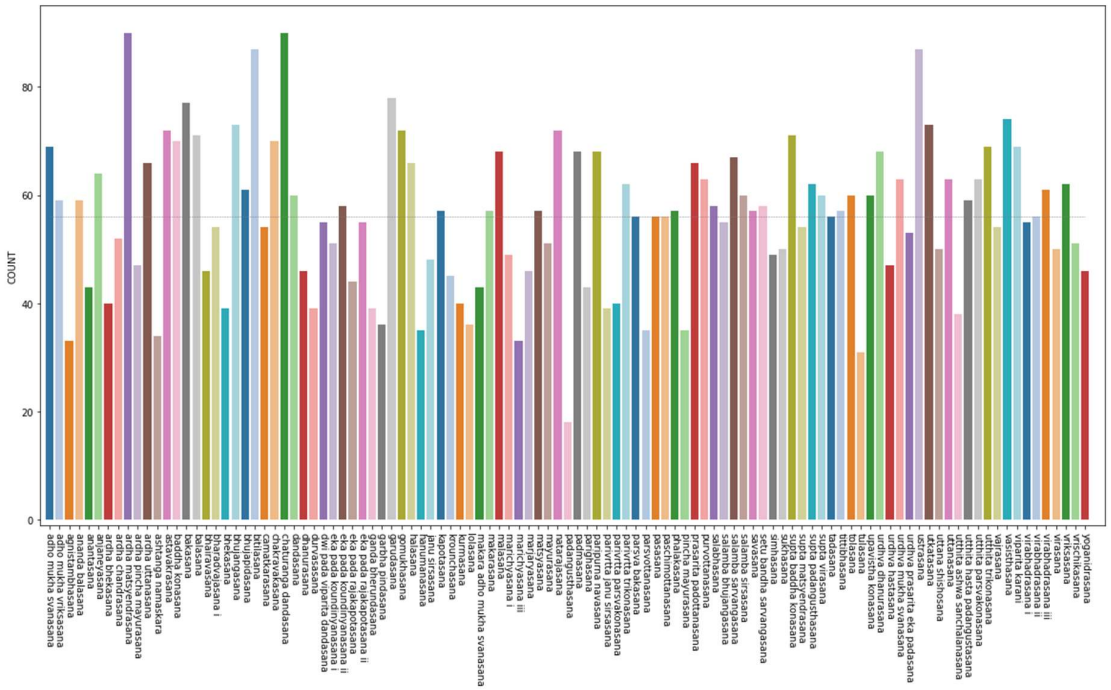

Figure 4: Distribution of data between classes. The figure shows the number of images within each class.

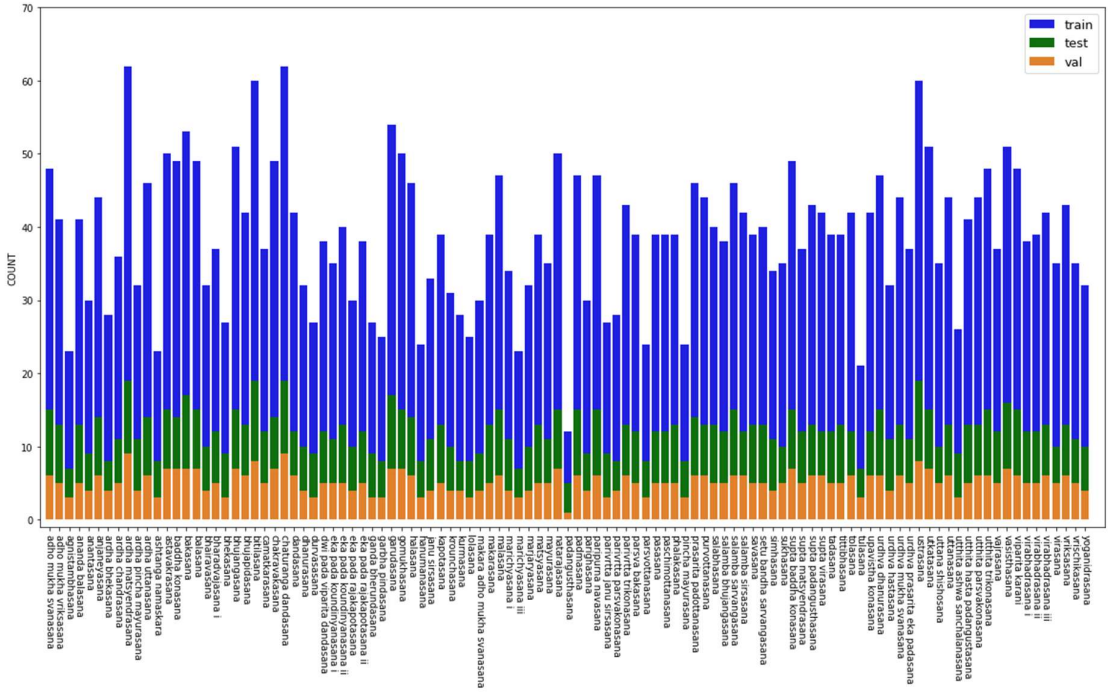

Figure 5: Distribution of training, test and validation data between yoga pose classes after data split. Blue for training set, green foe test set and orange for validation set.

Algorithms and Techniques

The model used is a supervised single-label image classification convolutional neural network algorithm.

The pre-trained VGG16 with Adadelta gradient descent optimization method and classifier layers of indexes 0, 3, and 6 with unfreeze weights was the model architecture chosen for this project. This model architecture was chosen after running smoke tests for two different supervised learning image classification neural network models and the two datasets. This part of the project was made in Google Colab to save AWS resources. The models tested were the pre-trained ResNet18 and VGG16 and the two datasets used were the original dataset (with mixed images mode) and the dataset with RGB-only images mode.

VGG16 model was added as described below:

```
def vgg():
    # download pretrained model
    model = models.vgg16(pretrained = True)

    # freeze all the pre-trained layers
    for p in model.parameters():
        p.requires_grad = False

    # get the numbers of input features in the last FC layer
    nin_features = model.classifier[6].in_features

    # replace the linear layers of the classifier
    model.classifier[0].requires_grad = True
    model.classifier[3].requires_grad = True

    # add the number of classes to out_features of last linear layer
    model.classifier[6] = nn.Sequential(
        nn.Linear(in_features = nin_features, out_features = 107),
        nn.LogSoftmax(dim = 1)
    )

    return model
```

```
def main(args):
    # Initialize the model
    model = vgg()

    # define loss
    loss_criterion = nn.CrossEntropyLoss()

    # define optimization criteria
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr) ### BEST  FIT!

    ## call the training function
    ...

    ## call test function
    ...

    # save the trained model
    save_model(model, args.model_dir)
```

The Figure 6 below shows an illustrates of the VGG16 architecture.

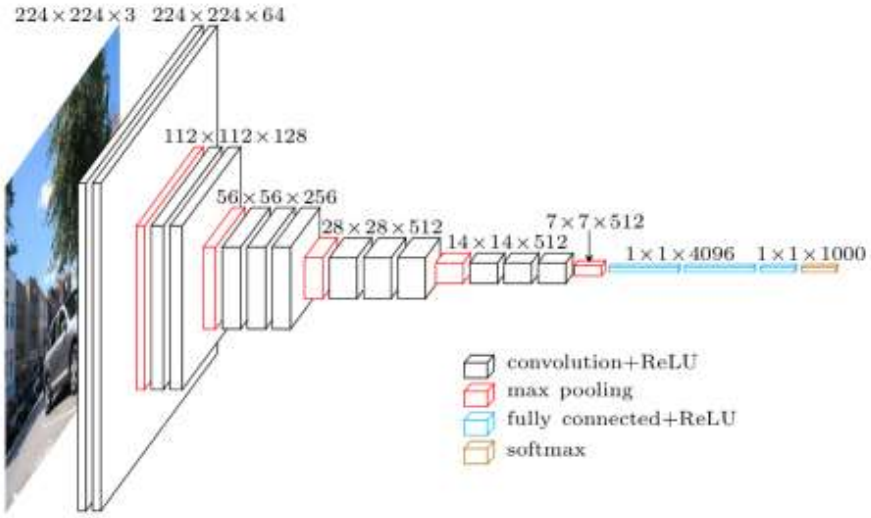

Figure 6: VGG16 architecture. Source [25, 26].

Benchmark

The solutions presented in the kaggle notebook "Classifying 107 classes of Yoga Asanas using state-of-the-art pre-trained EfficientNet model in PyTorch" [5] to classify the 107 classes of images of the yoga-pose-image-classification-dataset was chose as benchmark model. The intent was to compare resulting test loss and test accuracy of the benchmark model with those of the model used in this project.

The benchmark model used EfficientNet model architecture [8, 9] with classifier layer with unfreeze weights and Adam optimization method. It reached a test loss of 2.951 and test accuracy of 0.288 (29%) at 20 epochs.

Data Preprocessing

During data preprocess images file extensions were converted to .jpg, images file formats were converted to JPEG, and images mode were converted to 3-channels RGB mode. This process involved defining the methods for converting image mode to RGB mode, testing the conversion method in a random sample of images, visualizing the output of the conversion (Figure 7), converting the full dataset (convert all the images to RGB mode), and finally compressing the processed data as a zip file and uploading it back into Kaggle. This preprocessed dataset was made available at kaggle [21] for public access and further use during model development phase.

A method had to be created to convert images mode to 3-channels RGB mode. The use of images in RGB mode to finetune the pretrained model satisfy Torchvision Image Classification Models input image expectation [19].

Below is the output of data features checking after correcting for file extension, file format, and image mode:

```
1 file extensions found:
['jpg']

1 file formats found:
['JPEG']
```

```
1 image modes found:
['RGB']
{'RGB': 5994}
```

From this point the data was split into train/test/validation subsets and manifest files in *lst* format where created to be fed into model input channels of SageMaker Python SDK [14].

Data preprocessing only finishes during training. Before the data is passed to the training function the images are normalized to a shape of 3x224x224 to comply with Torchvision pre-trained models input image expectation [19]. The transformation used to normalize the images is given as follow:

```
normalize = transforms.Normalize(
        mean = [0.485, 0.456, 0.406],
        std = [0.229, 0.224, 0.225]
)
```

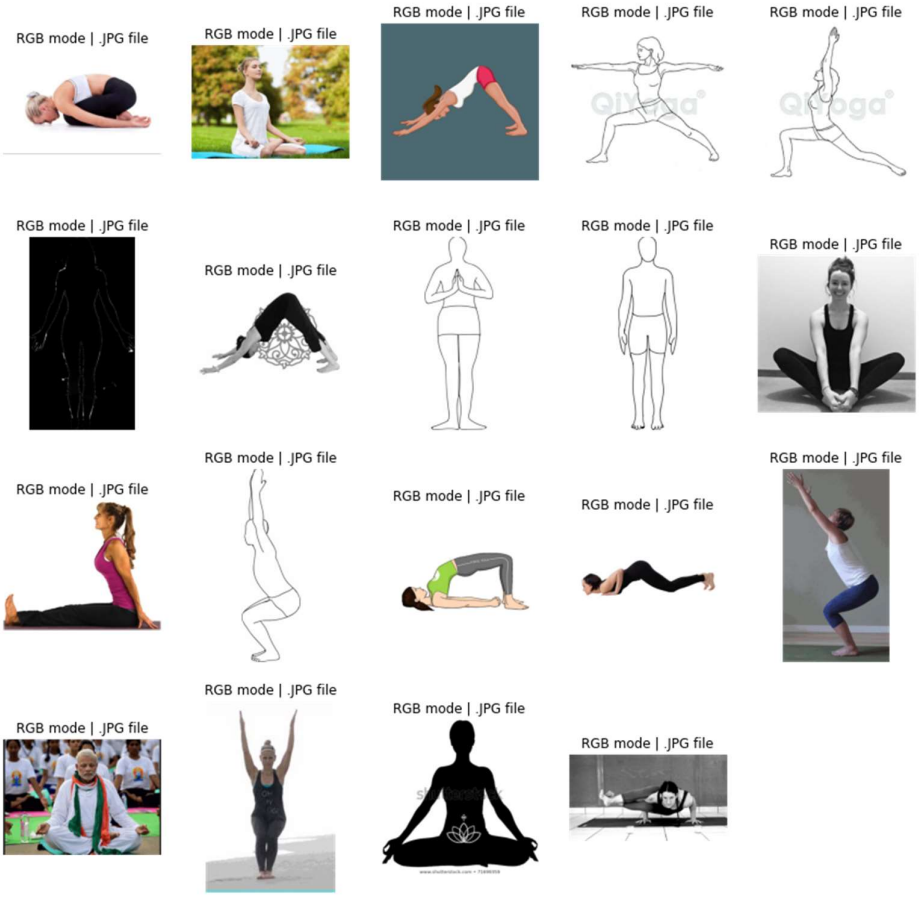

Figure 7: Figure shows a sample of images from kaggle "RGB-only-yoga-pose-dataset" with respective files extension and images mode. It illustrates how the images in this sample of images looked after image mode conversion. Note that all the images have RGB mode and .jpg file extension.

Implementation

Data issues:

Issues with images file extensions, file format, and images mode where fixed during data preprocessing. To comply with Torchvision Image Classification Models input image expectation [19] the images modes were converted to 3-channels RGB mode.

Here we describe the method used to convert the images mode to RGB mode. The method requires Pillow 9.2.0 and makes use of its Image Module [15, 16]. Different mode conversion operations are applied depending on the kind of mode to be converted to RGB:

1) The conversion from L to RGB mode is made using the Pillow function *Image.convert*.

```
img = img.convert('RGB')
```

2) The conversion from LA to RGB mode was inspired by the solution presented at stackoverflow: 'Python PIL remove every alpha channel completely' [20], and it is made in three steps:

first the LA mode is converted to RGBA mode using the Pillow method *Image.composite*

```
# convert to 'RGBA'
t = img.split()
# create an (white) image object with 'RGBA' mode
Lyr = Image.new(mode='RGBA', size=img.size, color=(255, 255, 255))
img = Image.composite(t[0], Lyr, t[1])
```

then RGBA is converted to P mode using the pillow function *Image.quantize*

```
# convert from 'RGBA' to 'P'
img = img.quantize(colors = 256)
```

and, finally from P mode to RGB mode using the Pillow function *Image.convert*.

```
# convert from 'P' to 'RGB'
img = img.convert('RGB')
```

3) The conversion from P mode to RGB mode is done by composing the P image channel with two RGBA images: 1) a RGBA mode mask image created with the data of the original (P) image, and 2) a RGB white image.

```
newData = []
# create a mask
for item in list(img.getdata()):
    newData.append(img2mask(item))
# 1) create a RGBA mode mask image by placing img data into a RGBA mode image
temp2 = Image.new(mode='RGBA', size=img.size, color=(255, 255, 255, 0))
temp.putdata(newData)
# 2) create a white image
temp2 = Image.new(mode='RGB', size=img.size, color=(255, 255, 255))
# 3) composite the two previous images with the original image channel
img = Image.composite(img.split()[0], temp2, temp)
```

The method of converting image mode to RGB was made publicly available in the github repository fsoaresantos/Yoga-pose-Image-Classification-Project (github.com) notebook

convert_image_mode2rgb.ipynb. This method focus on the images mode found in the dataset used in this project and can't be used to convert others types of image mode.

Training instances:

During staging the model was run with *ml.ml.4xlarge* instance type (CPU) and took 9434 seconds (2h 38min 58s) to run 30 epochs.

ml.ml.4xlarge instance was also used during model debugging and may be the cause of issues with time spent between steps taking longer than train and the eval phases displayed in the SageMaker Debugger Profiling Report.

The final model was run with *ml.p2.xlarge* instance type (GPU) and took circa 3973 seconds (1h 8min 46s) to run 55 epochs. However, due to resource limitations it was not possible to launch another training job.

Refinement

Choosing the model architecture:

The two pretrained model architectures (i.e., ResNet18 with the weights of its last fully connected layer unfreeze and VGG16 with the weights of its classifier layers of indexes 0, 3, and 6 unfreeze) were contrasted by comparing their respective test loss and test accuracy for a fixed set of hyperparameters. Simulations were run for value of epoch varying from 1 to 6.

A total of six smoke tests were run before chose the final model architecture. The scenarios tested where the following: ResNet18 with Adam optimization; VGG16 with SGD optimization method; and VGG16 with Adadelta optimization method. These three scenarios were tested with both datasets, i.e., 1) with the original dataset (with mixed images mode) and 2) with the RGB-only dataset.

Differences in running time and test accuracy were observed. VGG16 usually took much longer to run compared to ResNet18 (from 4 to 7 times longer). VGG16 test accuracy was higher compared to ResNet18. VGG16 test accuracy improved after replacing the SGD optimizer with Adadelta [1] (little architecture modification). Therefore, the VGG16 model architecture with Adadelta optimizer displayed the best accuracy. The VGG16 accuracy improved a little when trained with the RGB-only image mode dataset. The Figure 8 shows the result of the six experiments for different values of epoch.

Based on these experiments the pre-trained VGG16 with Adadelta gradient descent optimization method was the model architecture chosen for this project and the RGB-only image mode dataset was the dataset used to train the model.

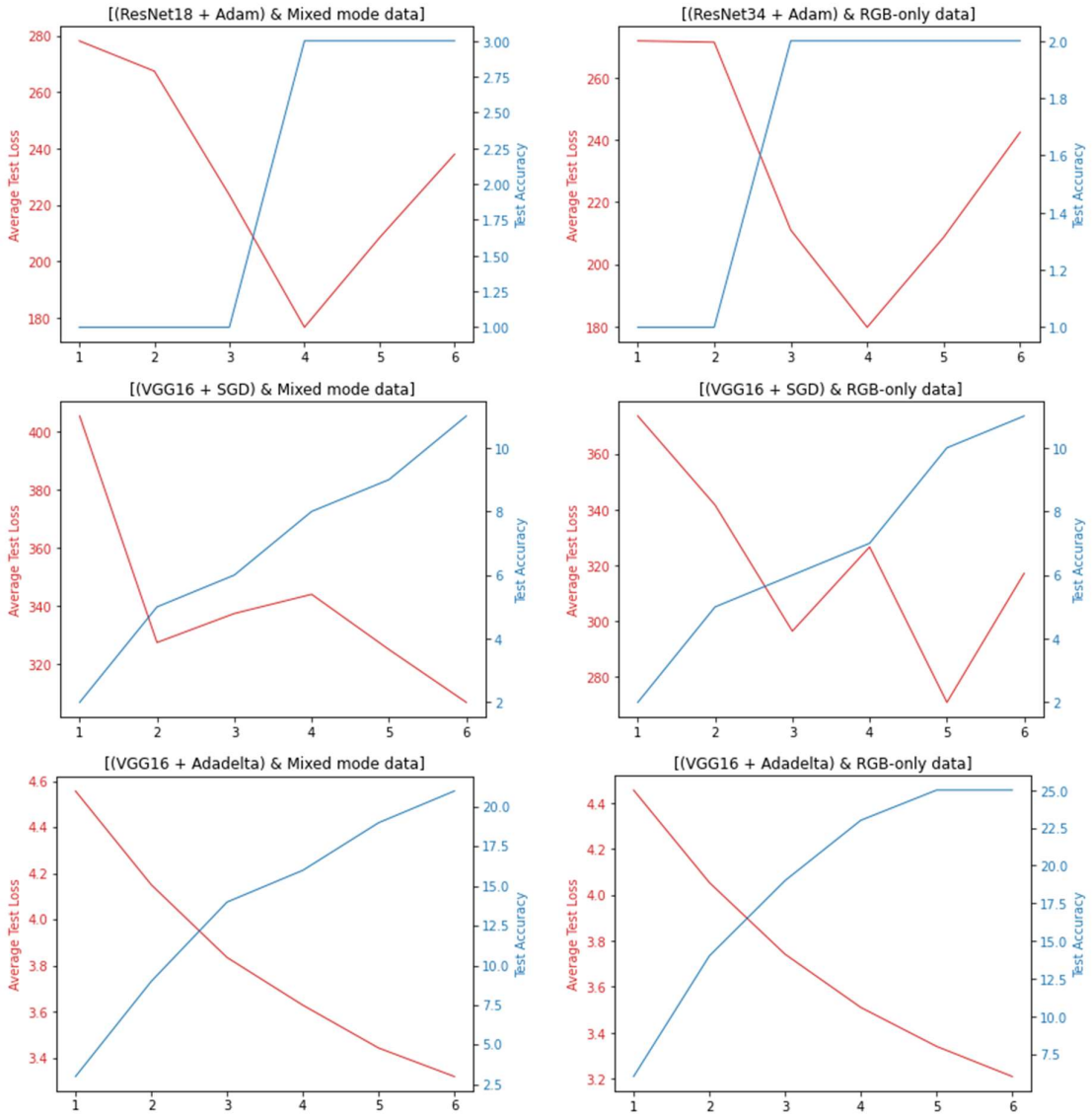

Figure 8: The figure shows the changes in the values of average test loss and test accuracy for a range of 6 epochs and for each one of the six scenarios tested before choosing the model architecture. Namely: ResNet18 + Adam optimization and dataset with mixed images mode; ResNet18 + Adam optimization and dataset with RGB-only images mode; VGG16 + SGD optimization method and dataset with mixed images mode; VGG16 + SGD optimization method and dataset with RGB-only images mode; VGG16 + Adadelta optimization method and dataset with mixed images mode; and VGG16 + Adadelta optimization method and dataset with RGB-only images mode.

Finetuning the model:

The model is finetuned in AWS Sagemaker using PyTorch as ML framework. Finetuning is performed in 4 steps for different ranges of the following hyperparameters: epoch, batch size, test batch size, and learning rate.

First an Hyperparameter Tuning Job is launched. Then a Warm Start Hyperparameter Tuning is performed to increase the number of hyperparameters values tuned. After this last step it was possible to narrow the range of hyperparameters to be tuned to a smaller interval which showed better performance. In addition, the value of the hyperparameter *epoch* was set to static and equal to 6 (to extract its influence). A second Hyperparameter Tuning Job is launched for this new set of hyperparameters ranges. The result allowed to exclude the *test_batch_size* from the set of hyperparameter to be tuned and set its value static and equal to 20. The range of values of the hyperparameter *batch_size* was narrowed based on the result of this hyperparameter tuning job. Finally, a second Warm Start Hyperparameter Tuning job is launched which allowed to find a good set of hyperparameters.

The Table 1 shows how the hyperparameters ranges were defined.

Set of hyperparameters range used to run the first Hyperparameter Tuning Job:

```
hyperparameter_ranges = {
        "lr": ContinuousParameter(0.005, 0.015,
scaling_type='Logarithmic'),
        "batch_size": CategoricalParameter([16, 20, 30, 40]),
        "test_batch_size": CategoricalParameter([10, 16, 20]),
        "epochs": CategoricalParameter([5, 10])
}
```

Set of hyperparameters range used to run the second Hyperparameter Tuning Job:

```
hyperparameters = {"epochs": "6"}
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.015, 0.125, scaling_type='Logarithmic'),
    "batch_size": CategoricalParameter([16, 20, 30]),
    "test_batch_size": CategoricalParameter([16, 20])
}
```

Set of hyperparameters range used to run the second Warm Start Hyperparameter Tuning job:

```
hyperparameters = {
    "epochs": "6",
    "test_batch_size": "20"
}
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.120, 0.125, scaling_type='Logarithmic'),
    "batch_size": CategoricalParameter([20, 30])
}
```

Table 1: Hyperparameter range definitions for subsequent hyperparameter tuning jobs.

The Table below (Table 2) shows the result of all hyperparameter tuning jobs ordered by final objective value (i.e., average test loss).

trainning job_id	batch_size	epochs	lr	test_batch_size	TrainingJobName	FinalObjectiveValue
25	"30"	"6"	0.121821	"20"	yoga-data-sec-warmst-221010-1656-002-1878c85f	0.155509
21	"30"	"6"	0.124203	"20"	yoga-data-sec-warmst-221010-1656-006-d5eee67b	0.155564
26	"30"	"6"	0.121777	"20"	yoga-data-sec-warmst-221010-1656-001-433b6b1a	0.15579
11	"30"	"6"	0.121623	"20"	yoga-data-221004-1441-010-b8df99a3	0.162172
24	"20"	"6"	0.120664	"20"	yoga-data-sec-warmst-221010-1656-003-75ddfa31	0.162871
23	"20"	"6"	0.12095	"20"	yoga-data-sec-warmst-221010-1656-004-d6a4f16e	0.163259
22	"20"	"6"	0.120222	"20"	yoga-data-sec-warmst-221010-1656-005-38b5ff5c	0.163851
13	"20"	"6"	0.125	"20"	yoga-data-221004-1441-008-c92f51c6	0.16745
12	"16"	"6"	0.120279	"20"	yoga-data-221004-1441-009-67309d49	0.17492
14	"30"	"6"	0.063256	"20"	yoga-data-221004-1441-007-a2c91766	0.179574
15	"16"	"6"	0.049801	"20"	yoga-data-221004-1441-006-31ed847a	0.187424
5	"16"	"10"	0.015	"20"	yoga-data-warmstart-221003-0705-006-1efd342b	0.195014
0	"16"	"10"	0.015	"20"	yoga-data-221001-1053-006-977a3a98	0.195545
20	"16"	"6"	0.017569	"20"	yoga-data-221004-1441-001-ddee9d6a	0.210159
19	"20"	"6"	0.108763	"16"	yoga-data-221004-1441-002-2daf2b77	0.213266
1	"30"	"10"	0.006869	"20"	yoga-data-221001-1053-001-1ea1055c	0.214853
16	"16"	"6"	0.099809	"16"	yoga-data-221004-1441-005-1df231ee	0.219874
10	"30"	"5"	0.009932	"20"	yoga-data-warmstart-221003-0705-001-1426c381	0.220627
17	"30"	"6"	0.056973	"16"	yoga-data-221004-1441-004-6901c8c0	0.22818
2	"20"	"10"	0.009764	"16"	yoga-data-221001-1053-004-3027d64b	0.256991
18	"20"	"6"	0.017917	"16"	yoga-data-221004-1441-003-03bb79e8	0.260718
3	"40"	"5"	0.005823	"16"	yoga-data-221001-1053-003-6453519c	0.284018
8	"16"	"10"	0.011565	"10"	yoga-data-warmstart-221003-0705-003-dd5cf26e	0.401997
7	"40"	"10"	0.005663	"10"	yoga-data-warmstart-221003-0705-004-6b11f9f7	0.440784
6	"40"	"10"	0.005105	"10"	yoga-data-warmstart-221003-0705-005-a7434cd7	0.44234
9	"30"	"5"	0.006317	"10"	yoga-data-warmstart-221003-0705-002-c110c11f	0.449626
1	"20"	"5"	0.006784	"10"	yoga-data-221001-1053-005-8844d49b	0.449771
4	"40"	"5"	0.007682	"10"	yoga-data-221001-1053-002-4f165926	0.449968

Table 2: Table of set of hyperparameters tuned (batch size, epoch, learning rate, and test batch size) and its respective final objective value (i.e., average test loss).

Model Evaluation and Validation

The training process was performed over 30 epochs and 55 epochs. The model achieved an average test loss of 0.128 and a test accuracy of 34.466% when trained for 30 epochs. Test accuracy did not improve when training for 55 epochs (average test loss of 0.129 and a test accuracy of 34.775%).

The trained model was validated using the images from validation set. For a sample of 100 images the model achieved **39%** of correct predictions. The result of these 100 model inferences is illustrated in Figure 9.

The table 3 below shows the resulting test loss and test accuracy for both this project model and the benchmark model (see “Benchmark” section above for information on benchmark model).

There is not enough data to verify if the difference between the two models is statistically significant because of limitations (GPU resource) to run several model training repetitions. Therefore, it was not possible to apply any statistical significance test to compare the two model’s metrics.

Benchmark model	This project model
`Test Loss: 2.950738` `Test Accuracy: 0.287829 (29%)`	`Test loss: 0.129022` `Test Accuracy: 0.351623 (35%)`
`Epoch: 20` `Train Loss: 1.618626`	`Epoch: 20` `Train Loss: 0.078499`

Table 3: Resulting test loss, test accuracy, and training loss for benchmark model in contrast to the model used in this project.

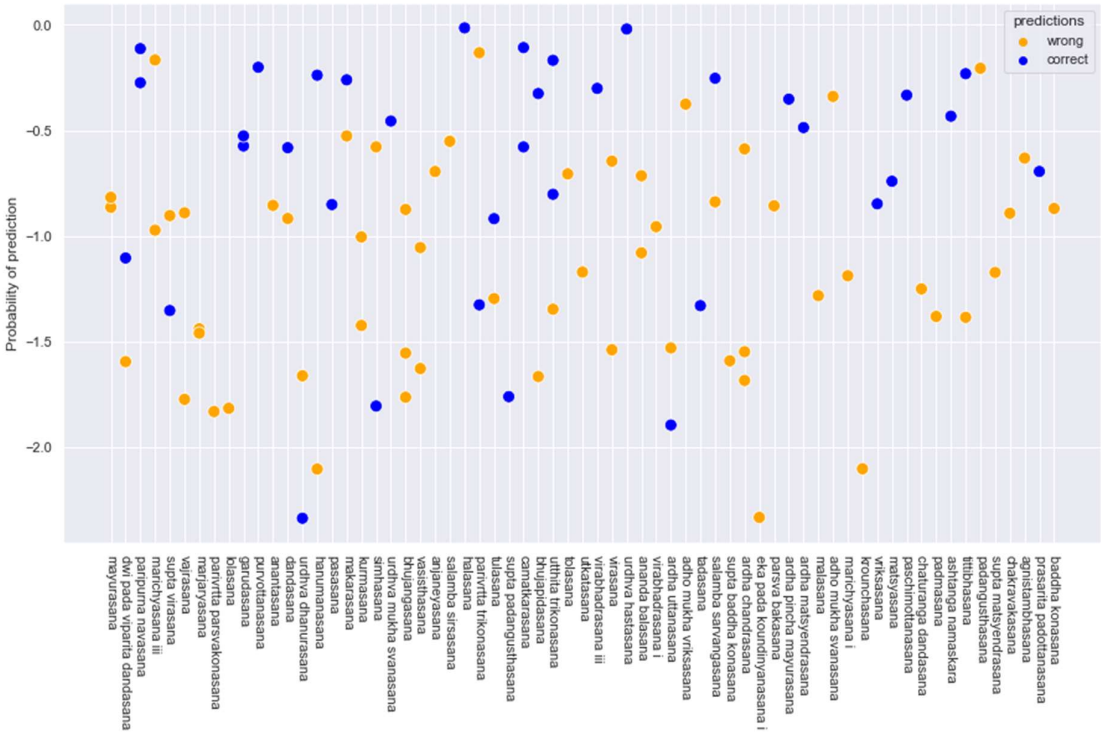

Figure 9: 100 inference points. The model did correct predictions 39% of the time (39 correct against 61 wrong)

To perform McNemar's test for comparing deep learning models it is necessary the existence of data about the two models' evaluation of the same instances [10]. The lack of this type of data together with the fact that the models may not have been trained on the same training data, even though both were trained on the same dataset, limited the use of McNemar's comparison test.

Figure 10 shows train and test loss data from the smoke test (before finetuning) for the model used in this project.

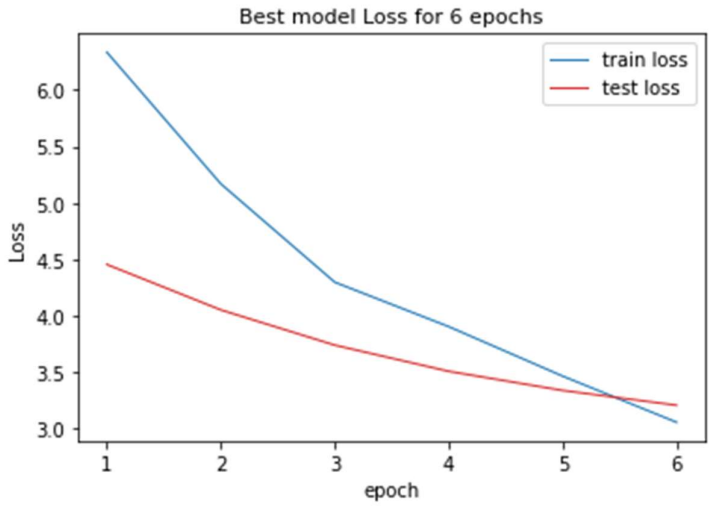

Figure 10: Best model train and test losses for 6 epochs.

Figure 11 shows one output of web application created during deployment to production phase to enable user interaction with the model endpoint.

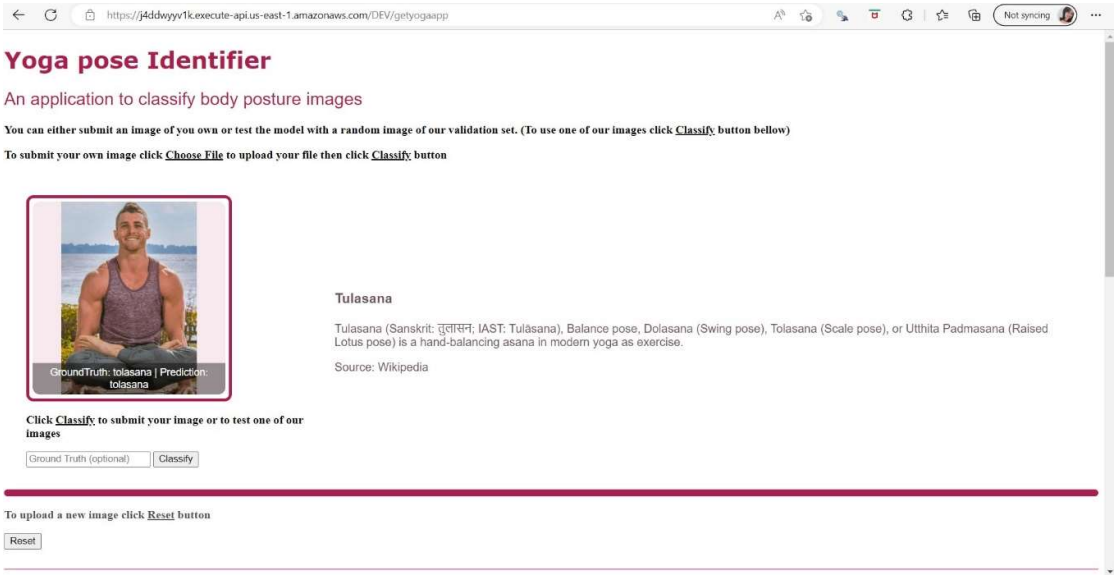

Figure 11: The figure shows an output of the webapp for a request to classify a random image of the validation dataset. This APP allows the user to use the model to classify an image of a yoga pose by submitting an image file or asking to use an image from the validation dataset. The APP also outputs the Wikipedia summary of the model prediction.

Justification

The model presented here is just an example of the use of Computer Vision on image recognition. The full potential of this model was not explored letting room for improvement. For instance, no measure was adopted in this project to mitigate issues caused by outliers and low represented classes of yoga pose in the dataset. More data could be collected focusing on low represented classes, and data augmentation could be used to try to give an uniform shape to the data distribution.

Although Pose Detection was not the focus of this work, a possible further improvement of this model presented here could consider the linkage (coupling) of the current image classification model to a pose estimator algorithm, like PoseNet, as done in [17] for instance.

Real-world use case has the potential to help on supervising of body positioning during physical exercises. AI can detect yoga pose and can potentially be used as an aid to accelerometers and gyroscope (which detects the speed and direction of movements) in fitness tracking applications, like Nintendo's Ring Fit Adventure, for instance, and others work out monitoring applications.

REFERENCES

[1] An Adaptive Learning Rate Method. https://paperswithcode.com/paper/adadelta-an-adaptive-learning-rate-method

[2] A Comprehensive Guide to Human Pose Estimation. https://www.v7labs.com/blog/human-pose-estimation-guide

[3] Boto3 documentation. SageMakerRuntime. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-runtime.html

[4] Call an Amazon SageMaker model endpoint using Amazon API Gateway and AWS Lambda. https://aws.amazon.com/blogs/machine-learning/call-an-amazon-sagemaker-model-endpoint-using-amazon-api-gateway-and-aws-lambda/

[5] Classifying 107 classes of Yoga Asanas using state-of-the-art pre-trained EfficientNet model in PyTorch. https://www.kaggle.com/code/kandhalkhandeka/efficientnet-pytorch-107-classes

[6] Color Modes Explained for Digital Image Processing in Python (PIL) – with Python Examples. https://holypython.com/python-pil-tutorial/color-modes-explained-for-digital-image-processing-in-python-pil/#1-mode

[7] Deploy Simple Web Applications in Lambda. https://aws-dojo.com/workshoplists/workshoplist47/

[8] EfficientNetV2: Faster, Smaller, and Higher Accuracy than Vision Transformers. https://towardsdatascience.com/efficientnetv2-faster-smaller-and-higher-accuracy-than-vision-transformers-98e23587bf04

[9] EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. https://paperswithcode.com/paper/efficientnet-rethinking-model-scaling-for

[10] How to Calculate McNemar's Test to Compare Two Machine Learning Classifiers. https://machinelearningmastery.com/mcnemars-test-for-machine-learning/

[11] How to Increase User Engagement in Online Workouts? Computer Vision in Fitness – Examples. https://neoteric.eu/blog/computer-vision-in-sports-and-fitness-examples/

[12] Image Classification for Autonomous Vehicles. https://github.com/hpatel530/Image-Classification-for-Autonomous-Vehicles-

[13] Image Classification Techniques. https://medium.com/analytics-vidhya/image-classification-techniques-83fd87011cac

[14] Image Classification – MXNet. Input/Output Interface for the Image Classification Algorithm. Train with Image Format. https://docs.aws.amazon.com/sagemaker/latest/dg/image-classification.html

[15] Image file formats. https://pillow.readthedocs.io/en/stable/handbook/image-file-formats.html#png

[16] Image Module. https://pillow.readthedocs.io/en/stable/reference/Image.html#generating-images

[17] ml5.js: Pose Classification with PoseNet and ml5.neuralNetwork(). https://www.youtube.com/watch?v=FYgYyq-xqAw

[18] Posture Detection using PoseNet with Real-time Deep Learning project. https://www.analyticsvidhya.com/blog/2021/09/posture-detection-using-posenet-with-real-time-deep-learning-project/

[19] PyTorch documentation. TORCHVISION.MODELS. https://pytorch.org/vision/0.11/models.html

[20] Python PIL remove every alpha channel completely. https://stackoverflow.com/questions/56815743/python-pil-remove-every-alpha-channel-completely

[21] RGB-only-yoga-pose-dataset. https://www.kaggle.com/datasets/franciscadossantos/rgb-only-yoga-pose-dataset

[22] Sagemaker Developer Guide: Deploy a Compiled Model Using SageMaker SDK. https://docs.aws.amazon.com/sagemaker/latest/dg/neo-deployment-hosting-services-sdk.html

[23] Use PyTorch with the SageMaker Python SDK. Deploy PyTorch Models. The SageMaker PyTorch Model Server. https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/using_pytorch.html#id3

[24] Using AWS Lambda with Amazon API Gateway. https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html#apigateway-types-transforms

[25] VGG in TensorFlow. https://www.cs.toronto.edu/~frossard/post/vgg16/

[26] VGG 16 Easiest Explanation. https://medium.com/nerd-for-tech/vgg-16-easiest-explanation-12453b599526

[27] Yoga Pose Detection and Classification Using Deep Learning. https://www.researchgate.net/publication/346659912_Yoga_Pose_Detection_and_Classification_Using_Deep_Learning

[28] Yoga Pose Detection Using Deep Learning Techniques. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3842656

[29] yoga-pose-image-classification-dataset. https://www.kaggle.com/datasets/shrutisaxena/yoga-pose-image-classification-dataset