Gesture Recognition Model Documentation

Base Model

Base model inspired by VGGNet16 model. All base model parameters remained the same as in VGGNet16 model except:

(a) Input Size: kept as 100x100 because some files in the training dataset provided had size 360x360, others had size 100x100; the latter was the lowest common denominator, hence chosen as such; reducing size might result in data loss; increasing size might add data that is not necessarily faithful to original input data

(b) CCN-2D vs. CNN-3D model: since problem at hand required analysis of multiple image frames / video, CCN-3D model (Conv3D model) was chosen; CNN filters / pooling layers were all 3D vs. 2D in VGGNet

(c) Optimzer chosen was Adam (with LR 0.001 initially) as performance was better than other optimisers with same initial Learning Rate (LR)

(d) Use of Batch Normalisation to regularise the model

Initial training was done in ablation mode with 100 of available 663 training samples.

Details of experiments / observations / measurements / decisions taken *below in Table 1 (PTO).*

Even after the "Final Model" (as per Table 1) was built, further experiments were conducted, including changing input data size, adding affine transformations (specifically rotations) along with cropping, cropping experiments, but these did not improve accuracy of the model

Note 1:

Following shorthands were used in Table 1

- VA: Validation Accuracy
- TA: Training Accuracy
- E: Epoch

Note 2:

Code for affine transformations not in Jupyter Notebook; hence adding to Appendix here

Table 1

Exp #	Observation	Statistics / Metric Name and Value	Decision Made & Explanation
1	Kernel Died		Batch Size changed from 512 -> 16; reduce memory load
2	Kernel Died		Remove one of the FC layers; repeat again; reduce memory load by reducing model parameters
3	Kernel Died		FC layer size change from 4096 -> 512; reduce memory load by reducing model parameters
4	Kernel Died		Remove last set of Conv layers x 2; reduce memory load by reducing model parameters
5	Kernel Died		Reduce num filters in last C layer: 256 -> 128 -> 64 -> 32; reduce memory load by reducing model parameters
6	Kernel Died		Reduce num filters in second C layer: 128 -> 64 -> 32; reduce memory load by reducing model parameters
7	Kernel Died		Reduce num filters in first C layer: 64 -> 32 -> 16; reduce memory load by reducing model parameters
8	Kernel Died		Remove last of layers in 3rd CNN set; reduce memory load by reducing model parameters
9	Increase Pooling Kernel Size		Num Neurons reduce: 2, 2, 2 -> 3, 3, 3; reduce memory load by reducing model parameters
10	Loss nan		Change optimiser to SGD; standard approach when seeing NaN loss
11	Loss nan		Change optimiser back to Adam, since no impact
12	Accuracy not improving with epochs	Accuracy in 20% range	Stop ablating, may be model needs more training data
13	Training accuracy improving but validation accuracy declining	Epoch 1: Val Acc 23% Epoch 2: Val Acc 23% Epoch 3: Val Acc 17%	Model is overfitting, use dropouts (0.2) after each set of layers
14	Training accuracy improving but validation accuracy declining	E1: VA 22% E2: VA 43% E7: VA 34%	Make the Conv set of layers single layer, model is overfitting
15	Training accuracy improved and plateaued at 95%+; training accuracy went up to 55% and then started dropping	21 epochs run; best VA: 55%	Model overfitting after 17 epochs; but VA needs to get better; increase num filters in 3rd CNN layer: 32 -> 64
16	Training accuracy growing, val accuracy increases but drops	9 epochs: TA: 36-94%, VA: E1-6: 17->52%; E7-9: 48->33%	Model is not generalising well; add one more layer of CNN abstraction with 8 filters; make it first layer
17	Training accuracy growing, val accuracy fluctuates without ramp	5 epochs: TA: 33-75%, VA: 16-36%	Model overfitting; add one more FC layer — hypothesis: will help abstract features and improve generalisability of model
18	Training accuracy growing, val accuracy fluctuates without ramp	7 epochs: TA: 33-87%, VA: 16-41%	Model overfitting; increase dropout in FC layers from 0.2 -> 0.5; add batch normalisation after each Conv Layer (currently BN only after pooling); pool_size in 3rd layer: 3,3,3-> 1,3,3
19	Training accuracy improving, val acc declining	7 epochs: TA: 25-56%; VA: 17-27%	Model overfitting; reduce num neutrons in FC layer 2: 512 -> 128
20	Training accuracy improving, val acc stable / not improving	17 epochs: TA: 25-70%, VA: 23-33%	Model overfitting; reduce num neutrons in FC layer 1 also: 512 -> 128
21	Training accuracy growing moderately, val accuracy fluctuates	19 epochs: TA 25-56%, VA: 25-43%	Model overfitting; reduce num_image_samples: 30 -> 8 + adjust pool layer first parameter; to improve training acc growth speed, change patience on LRonPlateau: 1 -> 5
Final Model	TA: 90%+	Best VA 67%	

Appendix: Code for Generator using Affine Transformations (Rotations)

```
def generator(source_path, folder_list, batch_size, ablation = 0, train_aug = False):

    #print('begin') # debug
    print('Source path = ' + source_path)
    print('batch size = ', batch_size)

    image_index = generate_image_index(num_image_samples)  # sample of images to be used for training

    #print('image_index = ', image_index) # debug

    #i = 0 # debug

    if train_aug == True:

        transform_deg = [-10, -5, 5, 10]

    while True:

        #print('Epoch Start') # debug

        if ablation == 0:

            randomised_folder_list = np.random.permutation(folder_list)

        else:

            randomised_folder_list = (np.random.permutation(folder_list))[0:ablation]
```

```
    num_full_size_batches = len(randomised_folder_list) // batch_size

    #print('num training samples: ', len(randomised_folder_list)) # debug
    #print('num_full_size_batches:' + str(num_full_size_batches)) # debug
    #print('total num image: ' + str(len(randomised_folder_list)*len(image_index))) # debug
    #print('len rand folder list: ' + str(len(randomised_folder_list))) # debug
    #print('len image index: ' + str(len(image_index))) # debug

    for batch in range(num_full_size_batches): # loop to generate dataset for all full-batches

        batch_data = np.zeros((batch_size, len(image_index), ideal_size_img[0], ideal_size_img[1], 3)) # initialise batch
        batch_labels = np.zeros((batch_size, 5)) # initialise one hot representation of labels

        #print(batch_data.shape) # debug
        #print(batch_labels.shape) # debug

        # make space for augmented data, if necessary

        if train_aug == True:

            batch_data = np.tile(batch_data, (len(transform_deg) + 1, 1, 1, 1, 1))
            batch_labels = np.tile(batch_labels, (len(transform_deg) + 1, 1))

            #print(batch_data.shape) # debug
            #print(batch_labels.shape) # debug

        for folder in range(batch_size): # loop to generate dataset for a single full-batch

            # read in file names
            image_file_names = os.listdir(source_path+'/'+\
```

```
                    randomised_folder_list[folder + (batch*batch_size)].split(';')[0])

            for idx, item in enumerate(image_index): # loop to read in each image in one batch

                # read in image within sequence (video)

                image = np.asarray(imageio.imread(source_path+'/'+ randomised_folder_list[folder +\
                        (batch*batch_size)].strip().split(';')[0]+'/'+image_file_names[item]).astype(np.float32))

                # retain copy for transformations, if any

                orig_image = image

                # crop using hyperparameters:"crop_top_pct", "crop_right_pct", "crop_bottom_pct",
"crop_left_pct"

                crop_start_top = int(np.floor(image.shape[0] * crop_top_pct))
                crop_end_right = int(image.shape[1] - np.ceil(image.shape[1] * crop_right_pct))
                crop_end_bottom = int(image.shape[0] - np.ceil(image.shape[0] * crop_bottom_pct))
                crop_start_left = int(np.floor(image.shape[1] * crop_left_pct))

                image = image[crop_start_top:crop_end_bottom, crop_start_left:crop_end_right, :]

                # resize image

                image = skimtr.resize(image, (ideal_size_img[0], ideal_size_img[1]))

                # normalise image using hyperparameter "normalisation_type" and feed into batch

                batch_data[folder, idx, :, :, 0] = normalised_image(image[:, :, 0], normalisation_type)
                batch_data[folder, idx, :, :, 1] = normalised_image(image[:, :, 1], normalisation_type)
```

```
            batch_data[folder, idx, :, :, 2] = normalised_image(image[:, :, 2], normalisation_type)

            # perform transformations

            if train_aug == True:

               for idz, deg in enumerate(transform_deg):

                  # rotate image by deg

                  rot_image = skimtr.rotate(orig_image, angle=deg)

                  # crop using hyperparameters:"crop_top_pct", "crop_right_pct", "crop_bottom_pct",
"crop_left_pct"

                  crop_start_top = int(np.floor(rot_image.shape[0] * crop_top_pct))
                  crop_end_right = int(rot_image.shape[1] - np.ceil(rot_image.shape[1] * crop_right_pct))
                  crop_end_bottom = int(rot_image.shape[0] - np.ceil(rot_image.shape[0] *
crop_bottom_pct))
                  crop_start_left = int(np.floor(rot_image.shape[1] * crop_left_pct))

                  rot_image = rot_image[crop_start_top:crop_end_bottom,
crop_start_left:crop_end_right, :]

                  # resize image

                  rot_image = skimtr.resize(rot_image, (ideal_size_img[0], ideal_size_img[1]))

                  # normalise image using hyperparameter "normalisation_type" and feed into batch

                  batch_data[folder + batch_size * (idz + 1), idx, :, :, 0] = \
                     normalised_image(rot_image[:, :, 0], normalisation_type)
```

```
                batch_data[folder + batch_size * (idz + 1), idx, :, :, 1] = \
                    normalised_image(rot_image[:, :, 1], normalisation_type)
                batch_data[folder + batch_size * (idz + 1), idx, :, :, 2] = \
                    normalised_image(rot_image[:, :, 2], normalisation_type)

            #pass # debug

        batch_labels[folder, int(randomised_folder_list[folder + (batch*batch_size)].strip().split(';')[2])]
= 1

        if train_aug == True:

            for idz, deg in enumerate(transform_deg):

                batch_labels[folder + idz, \
                    int(randomised_folder_list[folder + (batch*batch_size)].strip().split(';')[2])] = 1

        #i += 1 # debug

        #print(i) # debug

        yield batch_data, batch_labels

        #yield i # debug

    # code to generate dataset covering remaining folders

    num_remaining_input_seq = len(randomised_folder_list) - num_full_size_batches * batch_size

    batch_data = np.zeros((num_remaining_input_seq, len(image_index), ideal_size_img[0],
ideal_size_img[1], 3)) # initialise batch

    batch_labels = np.zeros((num_remaining_input_seq,5)) # initialise one hot representation of labels
```

```
        #print('num_remaining_input_seq:' + str(num_remaining_input_seq)) # debug

        if train_aug == True:

            batch_data = np.tile(batch_data, (len(transform_deg) + 1, 1, 1, 1, 1))
            batch_labels = np.tile(batch_labels, (len(transform_deg) + 1, 1))

        for idy, folder in enumerate(range(num_full_size_batches * batch_size,
len(randomised_folder_list))): # loop through remaining folders

            # read in file names
            image_file_names = os.listdir(source_path+'/'+ randomised_folder_list[folder].split(';')[0])

            for idx, item in enumerate(image_index): # loop to read in each image in one batch

                # read in image within sequence (video)

                image = np.asarray(imageio.imread(source_path+'/'+ randomised_folder_list[folder] \
                        .strip().split(';')[0]+'/'+image_file_names[item]).astype(np.float32))

                # retain copy for transformations, if any

                orig_image = image

                # crop using hyperparameters:"crop_top_pct", "crop_right_pct", "crop_bottom_pct",
"crop_left_pct"

                crop_start_top = int(np.floor(image.shape[0] * crop_top_pct))
                crop_end_right = int(image.shape[1] - np.ceil(image.shape[1] * crop_right_pct))
                crop_end_bottom = int(image.shape[0] - np.ceil(image.shape[0] * crop_bottom_pct))
                crop_start_left = int(np.floor(image.shape[1] * crop_left_pct))
```

```
            image = image[crop_start_top:crop_end_bottom, crop_start_left:crop_end_right, :]

            # resize image

            image = skimtr.resize(image, (ideal_size_img[0], ideal_size_img[1]))

            # normalise image using hyperparameter "normalisation_type" and feed into batch

            batch_data[idy, idx, :, :, 0] = normalised_image(image[:, :, 0], normalisation_type)
            batch_data[idy, idx, :, :, 1] = normalised_image(image[:, :, 1], normalisation_type)
            batch_data[idy, idx, :, :, 2] = normalised_image(image[:, :, 2], normalisation_type)

            if train_aug == True:

               for idz, deg in enumerate(transform_deg):

                  # rotate image by deg

                  rot_image = skimtr.rotate(orig_image, angle=deg)

                  # crop using hyperparameters:"crop_top_pct", "crop_right_pct", "crop_bottom_pct",
"crop_left_pct"

                  crop_start_top = int(np.floor(rot_image.shape[0] * crop_top_pct))
                  crop_end_right = int(rot_image.shape[1] - np.ceil(rot_image.shape[1] * crop_right_pct))
                  crop_end_bottom = int(rot_image.shape[0] - np.ceil(rot_image.shape[0] *
crop_bottom_pct))
                  crop_start_left = int(np.floor(rot_image.shape[1] * crop_left_pct))

                  rot_image = rot_image[crop_start_top:crop_end_bottom,
crop_start_left:crop_end_right, :]
```

```
                # resize image

                rot_image = skimtr.resize(rot_image, (ideal_size_img[0], ideal_size_img[1]))

                # normalise image using hyperparameter "normalisation_type" and feed into batch

                batch_data[idy + num_remaining_input_seq * (idz + 1), idx, :, :, 0] = \
                        normalised_image(rot_image[:, :, 0], normalisation_type)
                batch_data[idy + num_remaining_input_seq * (idz + 1), idx, :, :, 1] = \
                        normalised_image(rot_image[:, :, 1], normalisation_type)
                batch_data[idy + num_remaining_input_seq * (idz + 1), idx, :, :, 2] = \
                        normalised_image(rot_image[:, :, 2], normalisation_type)

            #pass # debug

        batch_labels[idy, int(randomised_folder_list[folder].strip().split(';')[2])] = 1

        if train_aug == True:

            for idz, deg in enumerate(transform_deg):

                batch_labels[idy + idz, int(randomised_folder_list[folder].strip().split(';')[2])] = 1

    #i += 1 # debug

    #print(i) # debug

    yield batch_data, batch_labels

    #break # debug
```

```
#yield i # debug


#print('Epoch End') # debug
```