PART A (I): Convolutional Neural Networks

Before we start, here are a few methods and variables which will be used.

Used for loading the data

```
def loadDataH5():
    with h5py.File('data1.h5','r') as hf:
        trainX = np.array(hf.get('trainX'))
        trainY = np.array(hf.get('trainY'))
        valX = np.array(hf.get('valX'))
        valY = np.array(hf.get('valY'))
        print (trainX.shape,trainY.shape)
        print (valX.shape,valY.shape)
    return trainX, trainY, valX, valY
```

Used for showing the graph

```
def showGraph(Histroy, epochs):
    # plot the training loss and accuracy
    plt.style.use("ggplot")
    plt.figure()
    plt.plot(np.arange(0, epochs), Histroy.history["loss"], label="train_loss")
    plt.plot(np.arange(0, epochs), Histroy.history["val_loss"], label="val_loss")
    plt.plot(np.arange(0, epochs), Histroy.history["accuracy"], label="train_acc")
    plt.plot(np.arange(0, epochs), Histroy.history["val_accuracy"], label="val_acc")
    plt.title("Training Loss and Accuracy")
    plt.xlabel("Epoch #")
    plt.ylabel("Loss/Accuracy")
    plt.legend()
    plt.show()
```

Used for Showing the graph

```
def trainModel(model, epochs, trainX, trainY, testX, testY):

    print (trainX.shape,trainY.shape)
    print (testX.shape,testY.shape)
    opt = tf.keras.optimizers.SGD(lr=0.01)
    print (model.summary())
    model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
        metrics=["accuracy"])
    print("Training network...")
    H = model.fit(trainX, trainY, validation_data=(testX, testY),
        batch_size=32, epochs=epochs)
    return H
```

```
input_shape = trainX.shape[1:]
classes = 17
epochs = 50
```

Model 1: BaselineCNN

```
baseline_CNN = tf.keras.Sequential()
baseline_CNN.add(tf.keras.layers.Conv2D (64, (3, 3), padding="same", input_shape=input_shape, activation='relu'))
baseline_CNN.add(MaxPooling2D(pool_size=(2,2)))
baseline_CNN.add(tf.keras.layers.Flatten())
baseline_CNN.add(Dense(64, activation = "relu")) # making the model fully connected
baseline_CNN.add(tf.keras.layers.Dense(classes, activation='softmax'))
```

Structure

```
(1020, 128, 128, 3) (1020,)
(340, 128, 128, 3) (340,)
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 128, 128, 64)      1792
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 64, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 262144)            0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16777280
_________________________________________________________________
dense_3 (Dense)              (None, 17)                1105
=================================================================
Total params: 16,780,177
Trainable params: 16,780,177
Non-trainable params: 0
```

Model 2: CNN2

```
CNN2 = tf.keras.Sequential()
CNN2.add(tf.keras.layers.Conv2D (64, (3, 3), padding="same", input_shape=input_shape, activation='relu'))
CNN2.add(MaxPooling2D(pool_size=(2,2)))

CNN2.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN2.add(MaxPooling2D(pool_size=(2,2)))

CNN2.add(tf.keras.layers.Flatten())
CNN2.add(Dense(256, activation = "relu")) # making the model fully connected
CNN2.add(tf.keras.layers.Dense(classes, activation='softmax'))
```

Structure

```
(1020, 128, 128, 3) (1020,)
(340, 128, 128, 3) (340,)
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 128, 128, 64)      1792
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 32, 32, 128)       0
_________________________________________________________________
flatten_2 (Flatten)          (None, 131072)            0
_________________________________________________________________
dense_4 (Dense)              (None, 256)               33554688
_________________________________________________________________
dense_5 (Dense)              (None, 17)                4369
=================================================================
Total params: 33,634,705
Trainable params: 33,634,705
Non-trainable params: 0
```

Model 3: CNN3

```
CNN3 = tf.keras.Sequential()
CNN3.add(tf.keras.layers.Conv2D (64, (3, 3), padding="same", input_shape=input_shape, activation='relu'))
CNN3.add(MaxPooling2D(pool_size=(2,2)))

CNN3.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN3.add(MaxPooling2D(pool_size=(2,2)))

CNN3.add(Conv2D(filters = 256, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN3.add(MaxPooling2D(pool_size=(2,2)))


CNN3.add(tf.keras.layers.Flatten())
CNN3.add(Dense(512, activation = "relu")) # making the model fully connected
CNN3.add(tf.keras.layers.Dense(classes, activation='softmax'))
```

```
(1020, 128, 128, 3) (1020,)
(340, 128, 128, 3) (340,)
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_4 (Conv2D)            (None, 128, 128, 64)      1792      
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 64, 64, 64)        0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 64, 64, 128)       73856     
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 32, 32, 128)       0         
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 256)       295168    
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 16, 16, 256)       0         
_________________________________________________________________
flatten_3 (Flatten)          (None, 65536)             0         
_________________________________________________________________
dense_6 (Dense)              (None, 512)               33554944  
_________________________________________________________________
dense_7 (Dense)              (None, 17)                8721      
=================================================================
Total params: 33,934,481
Trainable params: 33,934,481
Non-trainable params: 0
```

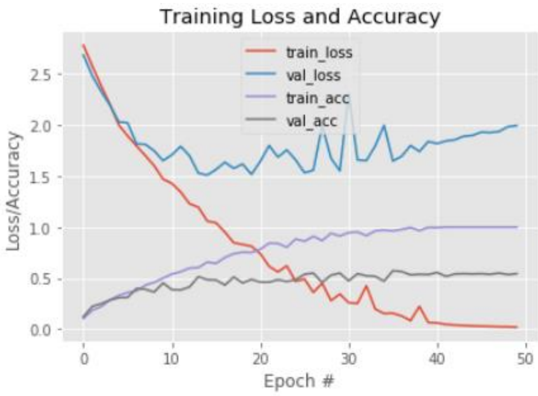
Model 4: CNN4

```
CNN4 = tf.keras.Sequential()
CNN4.add(tf.keras.layers.Conv2D (64, (3, 3), padding="same", input_shape=input_shape, activation='relu'))
CNN4.add(MaxPooling2D(pool_size=(2,2)))

CNN4.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN4.add(MaxPooling2D(pool_size=(2,2)))

CNN4.add(Conv2D(filters = 256, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN4.add(MaxPooling2D(pool_size=(2,2)))

CNN4.add(Conv2D(filters = 512, kernel_size = (3,3),padding = 'Same',activation ='relu'))
CNN4.add(MaxPooling2D(pool_size=(2,2)))

CNN4.add(tf.keras.layers.Flatten())
CNN4.add(Dense(1024, activation = "relu")) # making the model fully connected
CNN4.add(tf.keras.layers.Dense(classes, activation='softmax'))
```

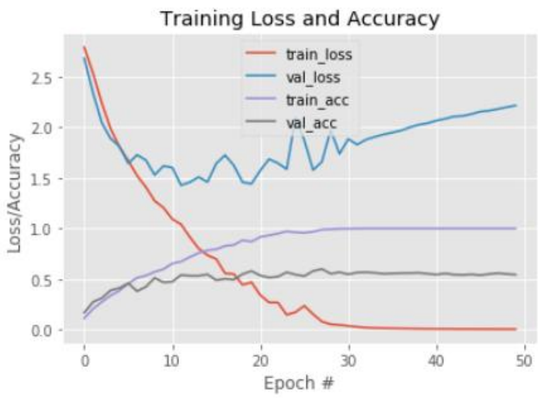
```
history = trainModel(CNN4, epochs, trainX, trainY, valX, valY)
showGraph(history, epochs)
```

Structure

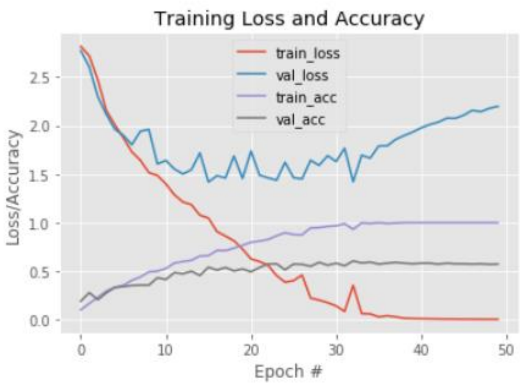
```
(1020, 128, 128, 3) (1020,)
(340, 128, 128, 3) (340,)
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 128, 128, 64)      1792
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 64, 64, 128)       73856
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 256)       295168
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 512)       1180160
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 8, 8, 512)         0
_________________________________________________________________
flatten_4 (Flatten)          (None, 32768)             0
_________________________________________________________________
dense_8 (Dense)              (None, 1024)              33555456
_________________________________________________________________
dense_9 (Dense)              (None, 17)                17425
=================================================================
Total params: 35,123,857
Trainable params: 35,123,857
Non-trainable params: 0
```



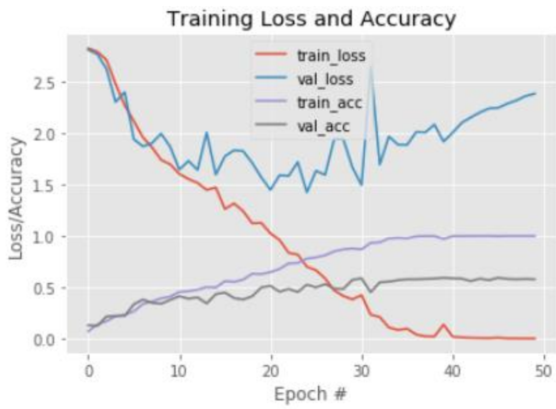
Baseline CNN



CNN 2 (2 Layers)



CNN 3 (3 Layers)



CNN 4 (4 Layers)

	Baseline Layer	CNN2	CNN3	CNN4
Overfitting epoch	9	6	6	10
Noise from epoch on validation loss	9	6	6	6
Noise graphs	Yes	Yes	Yes	Yes
Best train accuracy (Epoch)	42 (100%)	33 (100%)	36 (100%)	43 (100%)
Reached 100% train accuracy	Yes	Yes	Yes	Yes
Best validation accuracy (Epoch/percent)	36 epoch / 57.35%	28 epochs / 60%	33 epochs/ 60.59%	49 epochs/ 60.88 %
Reached 100% train accuracy	No	No	No	No
Increasing trend in validation loss	Yes	Yes	Yes	Yes
Increasing trend in validation loss epoch	11	9	13	11
Train accuracy at the overfitting epoch	54.51%	48.73%	38.84%	43.04%
Validation accuracy at the overfitting epoch	39.71%	46.18%	37.94%	35.88%

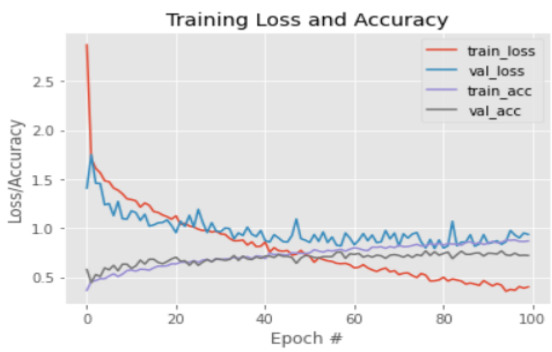
Observations

- Every model is overfitting, due to lack of data
- Adding more layers will not help to improve the performance of the model
- Train accuracy reaches to 100% but validation accuracy doesn't go above 60%
- Based on the accuracy at the epoch when the model starts to overfit. CNN 2 (2 layers) is best among these baseline models.

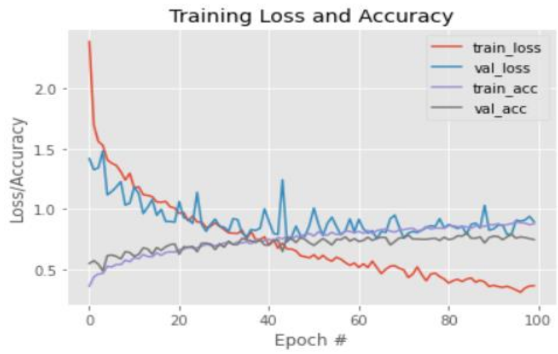
Question: Investigate the implementation of data augmentation techniques for two of the above models (please select the two deepest models). In your report describe the impact (if any), of applying data augmentation on these models.

Configuration on both networks (CNN with 3 and 4 Layers)

```
train_datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

train_datagen.fit(trainX)
```



Graph (3-layer CNN)



Graph (4-layer CNN)

	CNN3	CNN4
Overfitting epoch	40	35
Noise from epoch on validation loss	less	more
Noisy graphs	yes	yes
Best train accuracy (Epoch/percent)	97/87.94%	96/89.41%
Best validation accuracy (Epoch/percent)	95/76.76%	85/80.29%
An increasing trend in validation loss	No clear loss increase trend	Slight increasing loss trend
Increasing trend in validation loss epoch	NA	47
Train accuracy at the overfitting epoch	71.27%	71.86%
Validation accuracy at the overfitting epoch	69.12%	75.88%

Observations

- Defiantly by making use of data augmentation, now the models are performing much better.
- With a deeper network (4 later), there is some evidence of over-fitting but with our existing mild level data-augmentation, CNN 4 (4 layers) is performing better.

Question: How do you explain the impact of data augmentation? Does the selection of methods used as part of your data augmentation (such as cropping, flipping, etc) influence accuracy?

More aggressive data augmentation parameters

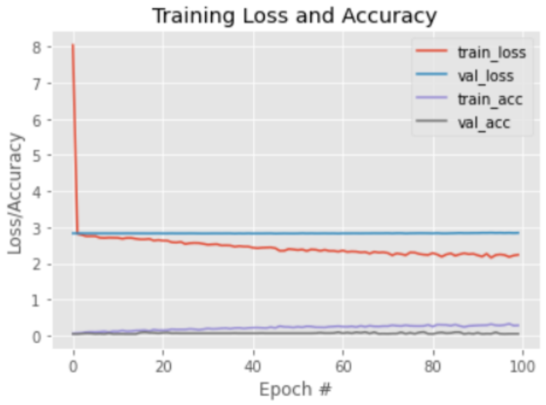
Configuration

```
train_datagen = ImageDataGenerator(rotation_range=90,
                                   width_shift_range=[0.1,0.5],
                                   height_shift_range=[0.1,0.5],
                                   shear_range=0.2,
                                   zoom_range=[0.1,0.5],
                                   horizontal_flip=True,
                                   brightness_range=[0.2,0.5],
                                   fill_mode="nearest")
train_datagen.fit(trainX)
```

```
history = trainModelAug(CNN4, train_datagen, 100, trainX, trainY, testX, testY)
showGraph(history, epochs)
```

Aggressive data augmentation on the CNNs

Aggressive data augmentation the CNN 3



Aggressive data augmentation the CNN 4

Observations

- CNN with 3 layers, just doesn't train. Train accuracy: 63.7%, Validation accuracy: 44.1%, Train loss: 2.8321, validation loss: 2.8403 through-out the training process.
- Aggressive data augmentation makes the network too complex to train
- CNN with 4 layer does train with reducing loss for train data but validation loss get stagnant at 2.83**.
- CNN with 4 Layer reached best 32.84% train accuracy on 98th epoch and best validation accuracy of just 9.4% at 95th epoch.
- Deeper network performing better

Note: We defiantly need to make data augmentation subtle, and check for the best possible combination to parameters.

Removed few data augmentation parameters on the deepest model

Configuration

```
train_datagen = ImageDataGenerator(rotation_range=20,
                                   shear_range=0.2,
                                   zoom_range=[0.1,1],
                                   horizontal_flip=True,
                                   brightness_range=[0.2,1.0],
                                   fill_mode="nearest")
train_datagen.fit(trainX)
```

```
history = trainModelAug(CNN4, train_datagen, 100, trainX, trainY, testX, testY)
showGraph(history, epochs)
```

Graph of CNN with 4 layers

	CNN4
Overfitting	No clear signs
Noisy graphs	Yes
Best train accuracy (Epoch/percent)	100/85.98%
Best validation accuracy (Epoch/percent)	98/10.88%
Increasing trend in validation loss	No

Observations:

- This configuration is performing poorly
- Aggressive data augmentation has shown adverse effect on the performance of the model

Making data augmentation more subtle ¶

```
train_datagen = ImageDataGenerator(rotation_range=45,
                                   horizontal_flip=True,
                                   brightness_range=[0.2,1.0])
train_datagen.fit(trainX)
```

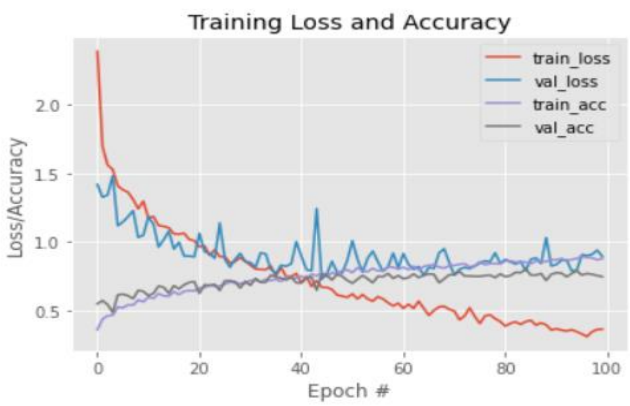
```
history = trainModelAug(CNN4, train_datagen, 100, trainX, trainY, testX, testY)
showGraph(history, epochs)
```

	CNN4
Overfitting	Yes
Noisy graphs	Yes
Best train accuracy (Epoch/percent)	91/100%
Best validation accuracy (Epoch/percent)	91/31.18%
overfitting epoch	1

The Best performing model with data augmentation

Configuration

```
train_datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

train_datagen.fit(trainX)
```



Graph (4-layer CNN)

	CNN4
Overfitting epoch	35
Noise from epoch on validation loss	more
Noise graphs	yes
Best train accuracy (Epoch/percent)	96/89.41%
Best validation accuracy (Epoch/percent)	85/80.29%
Increasing trend in validation loss	Slight increasing loss trend
Increasing trend in validation loss epoch	47
Train accuracy at the overfitting epoch	71.86%
Validation accuracy at the overfitting epoch	75.88%

Part A (ii): Ensemble technique

Ensemble technique 1 (Model Averaging on the different model structure)

Question: Build a CNN containing a maximum of 10 base learners.

Base Learner 1

```
def model1(classes, model_input):


    model = Sequential()

    model.add(Conv2D(filters = 16, kernel_size = (3,3),padding = 'Same',activation ='relu', input_shape = model_input))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters = 32, kernel_size = (3,3),padding = 'Same',activation ='relu'))

    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Flatten())
    model.add(Dense(32, activation = "relu"))
    model.add(Dense(classes, activation = "softmax"))
    return model
```

Base Learner 2

```
def model2(class_label,model_input):


    model = Sequential()

    model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'Same',activation ='relu', input_shape = model_input))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters = 32, kernel_size = (3,3),padding = 'Same',activation ='relu'))

    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =128, kernel_size = (1,1),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =32, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =96, kernel_size = (1,1),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Flatten())
    model.add(Dense(64, activation = "relu"))
    model.add(Dropout(0.4))
    model.add(Dense(class_label, activation = "softmax"))
    return model
```

Base Learner 3

```
def model3(class_label,model_input):


    model = Sequential()

    model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'Same',activation ='relu', input_shape = model_input))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',activation ='relu'))

    model.add(Conv2D(filters =32, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))


    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =512, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(128, activation='tanh'))
    model.add(Dropout(0.5))
    model.add(Dense(class_label, activation = "softmax"))
    return model
```

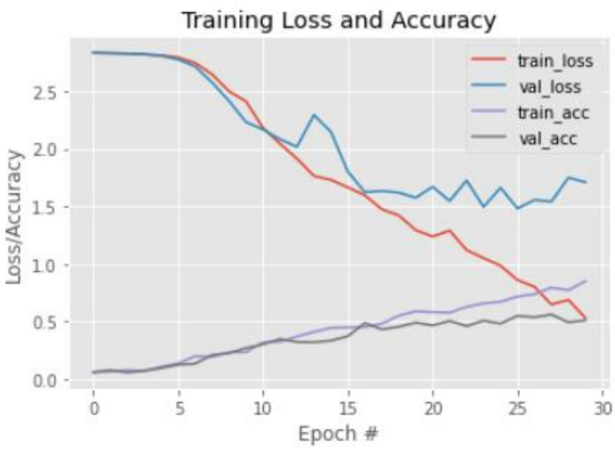
Base Learner 4

```
def model4(class_label,model_input):
    model = Sequential()

    model.add(Conv2D(filters = 16, kernel_size = (5,5),padding = 'Same',activation ='relu', input_shape = model_input))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters = 32, kernel_size = (1,1),padding = 'Same',activation ='relu'))

    model.add(Conv2D(filters =64, kernel_size = (1,1),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =32, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =64, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))


    model.add(Conv2D(filters =96, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =128, kernel_size = (3,3),padding = 'Same',activation ='relu'))
    model.add(Conv2D(filters =512, kernel_size = (3,3),padding = 'Same',activation ='relu'))

    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(256, activation = "relu"))
    model.add(Dropout(0.5))
    model.add(Dense(class_label, activation = "softmax"))
    return model
```

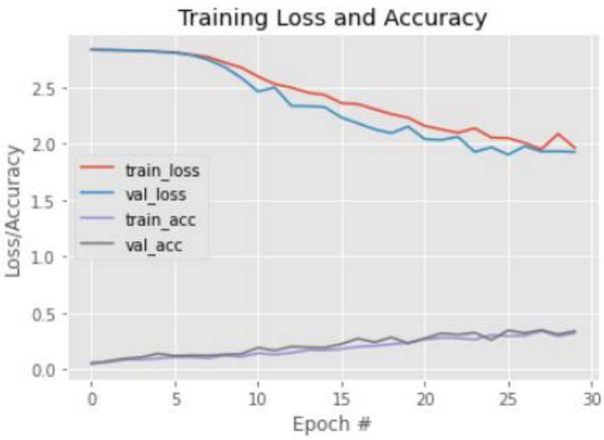
Supported methods

```python
def train_model(model,fileName,epochs):
    #trainX, trainY, valX, valY = LoadDataH5()
    model.compile(optimizer=SGD(lr=0.01), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    filepath = str(fileName)+".h5"
    checkpoint = ModelCheckpoint(filepath, monitor="val_loss", mode="min", save_best_only=True, verbose=1)
    history = model.fit(x = trainX, y = trainY, batch_size = 32, epochs = epochs,callbacks=[checkpoint], validation_data = (valX, valY))
    print(history)
    showGraph(history,epochs)
    return str(fileName)+".h5"
```

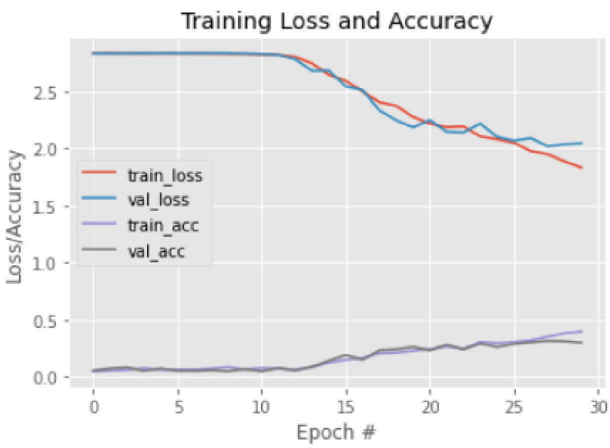
```python
def ensemble_prediction(models):
    predictions = []
    for model in models:
        predictions.append(model.predict_proba(valX))
    predict_Avg= np.array(predictions)
    result=np.argmax(np.sum(predict_Avg/len(models), axis=0),axis=1)
    score = accuracy_score(valY, result, normalize=True)
    print("Ensemble accuracy Score: ",score)
    return score
```



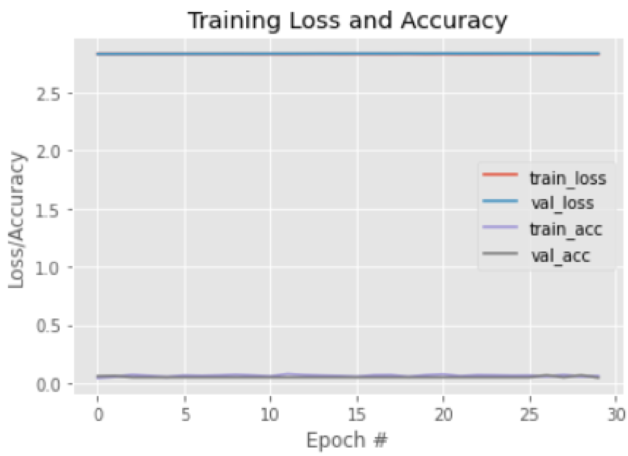
Model 1



Model 2



Model 2



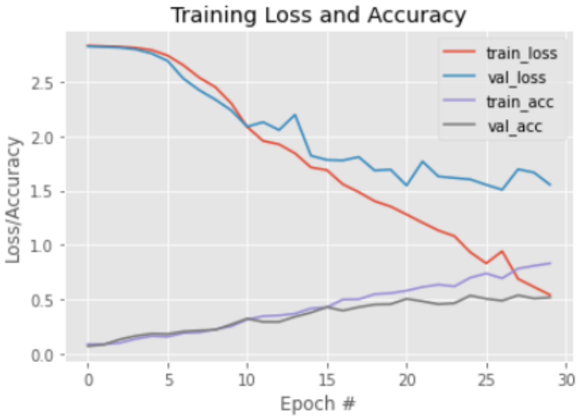
Model 4

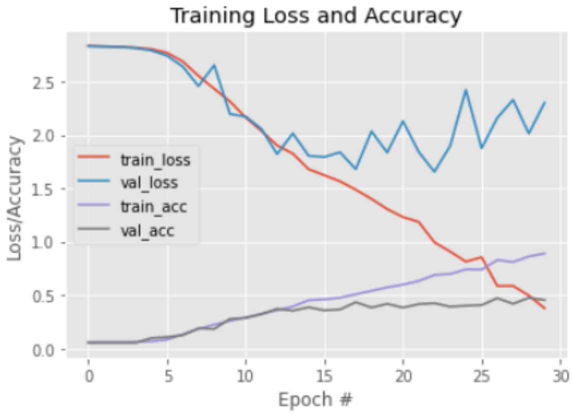
Comparisons

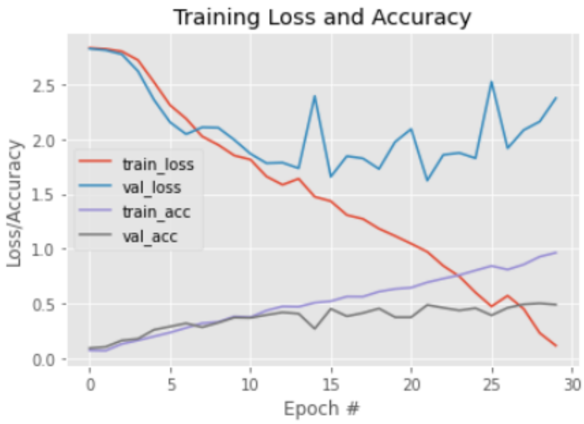
	Model1	Model2	Model3	Model4	Ensemble
Best train accuracy (Epoch/percent)	30/85.10%	28/33.92%	30/39.61%	30/5.69	54.70%
Best validation accuracy (Epoch/percent)	26/56.18	28/34%	28/31.47%	29/6.47	

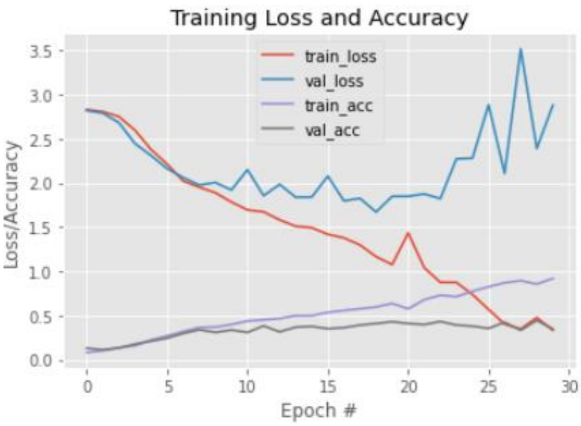
Ensemble technique 2 (Model Averaging on same model structure – 10 models with random initialization)

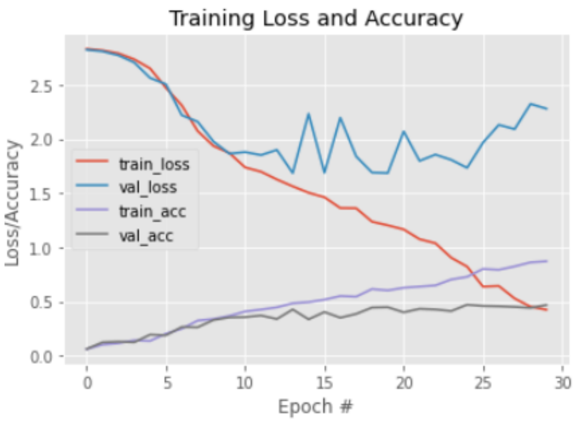
```
trainX, trainY, valX, valY = loadDataH5()
classes = 17
epochs = 30
models_list = []
score_list = []
input_shape = trainX.shape[1:]

#************ Model 1 *****************
for i in range(10):
    print("classes",classes)
    print("Inpur shape",input_shape)
    models_list.append(model1(classes, input_shape))
    #print(model1.summary())
    model1_weight_file = train_model(models_list[i], fileName='model1_take'+str(i),epochs=epochs)
    models_list[i].load_weights(model1_weight_file)
    #models_list.append(model1)
    result_1 = evaluate(models_list[i])
    score_list.append(result_1)

ensembleScore = ensemble_prediction(models_list)
```

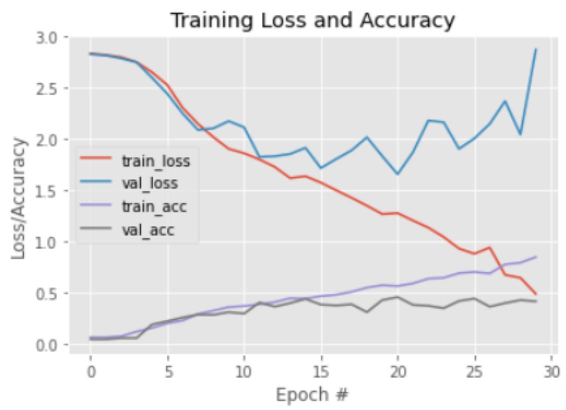


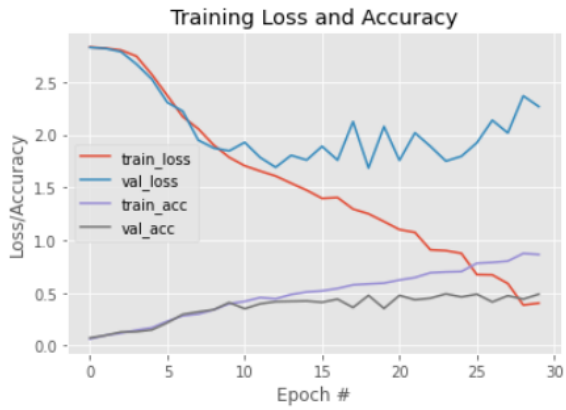


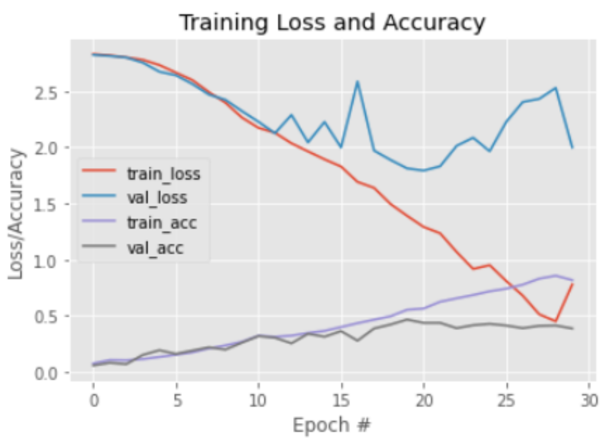


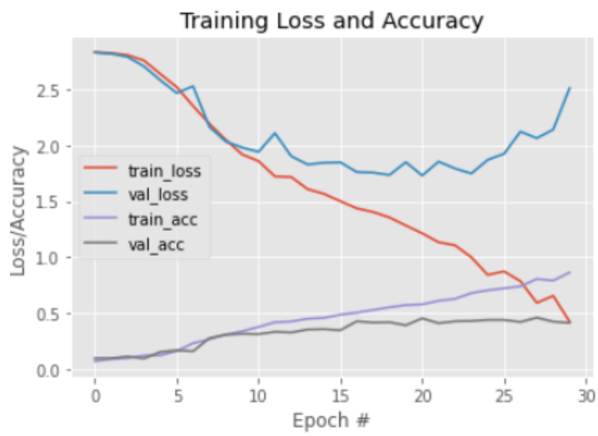








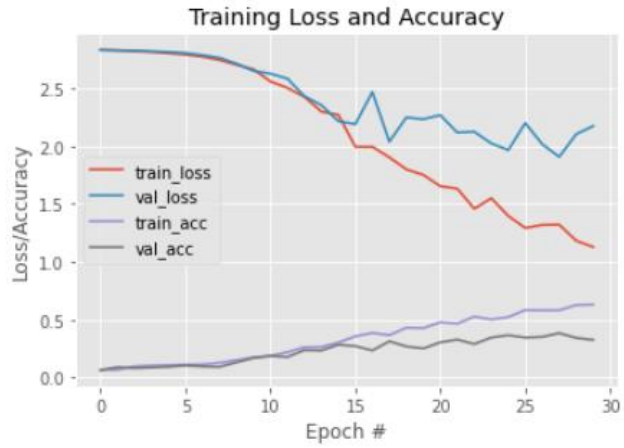


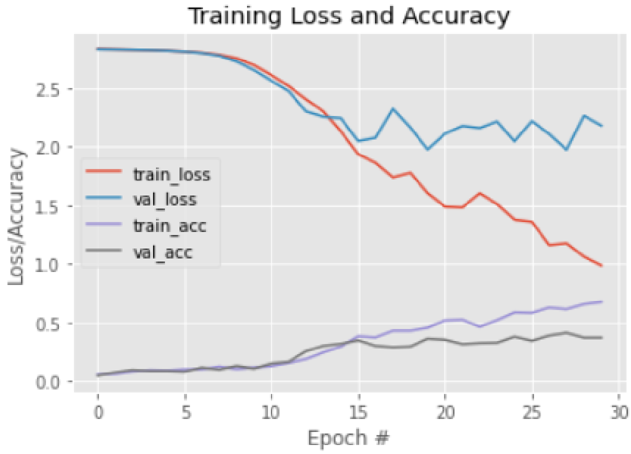


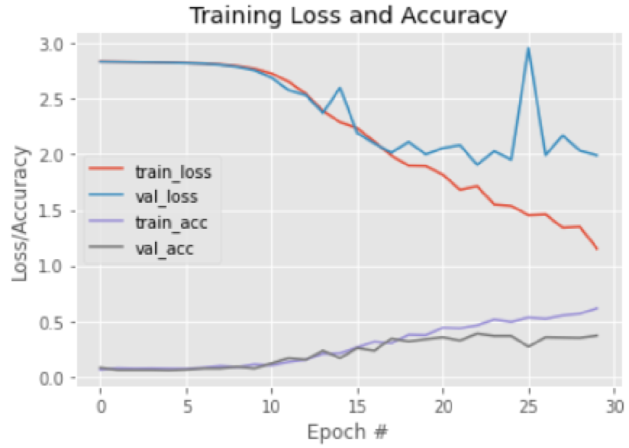
Model No	1	2	3	4	5	6	7	8	9	10	Ensemble
Best train accuracy (Epoch/percent)	**30/83.13%**	**30/88.82%**	**30/95.69%**	**30/96.47%**	**30/91.86%**	**30/87.06%**	**30/84.61%**	**30/86.47%**	**29/85.59%**	**30/86.47%**	**NA**
Best validation accuracy (Epoch/percent)	**28/53.82%**	**27/47.35%**	**26/48.24%**	**29/50.29%**	**29/44.41%**	**30/46.76%**	**26/44.12%**	**30/48.82%**	**20/43.53%**	**28/46.18%**	**51.57%**

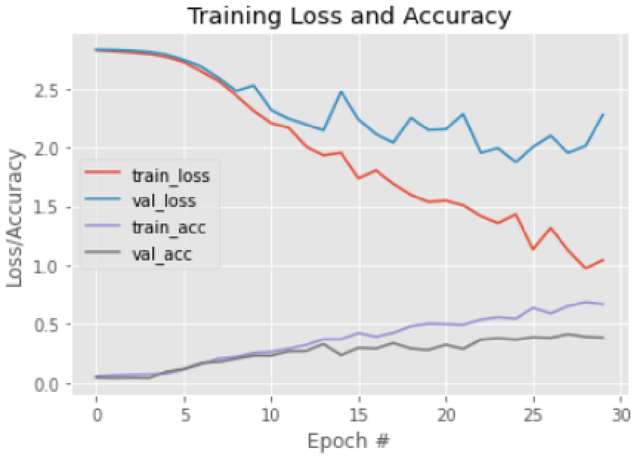
Ensemble technique 3 (Random Training Subset Ensemble)

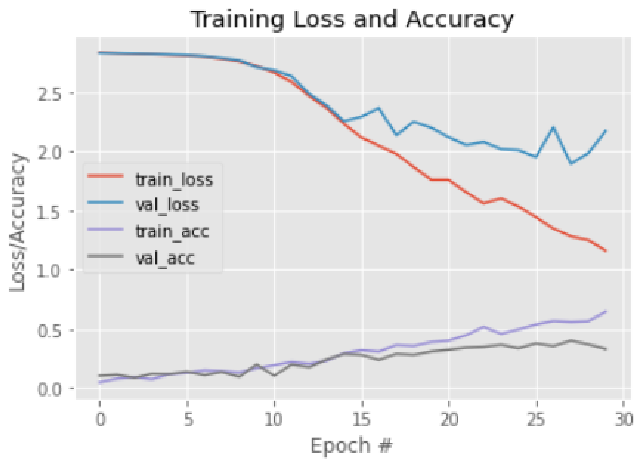
```
trainX, trainY, valX, valY = loadDataH5()
size = trainX.shape[0]
classes = 17
epochs = 30
models_list = []
score_list = []
input_shape = trainX.shape[1:]

#************ Model 1 *****************
for i in range(10):
    random_list = random.sample(range(size), int(size*0.5))
    newTrainX = trainX[random_list]
    newTrainY = trainY[random_list]
    models_list.append(model1(classes, input_shape))
    model1_weight_file = train_model(models_list[i], fileName='model1_take'+str(i),epochs=epochs, trainX=newTrainX, trainY=newTrainY)
    models_list[i].load_weights(model1_weight_file)
    result_1 = evaluate(models_list[i])
    score_list.append(result_1)

ensembleScore = ensemble_prediction(models_list)
```

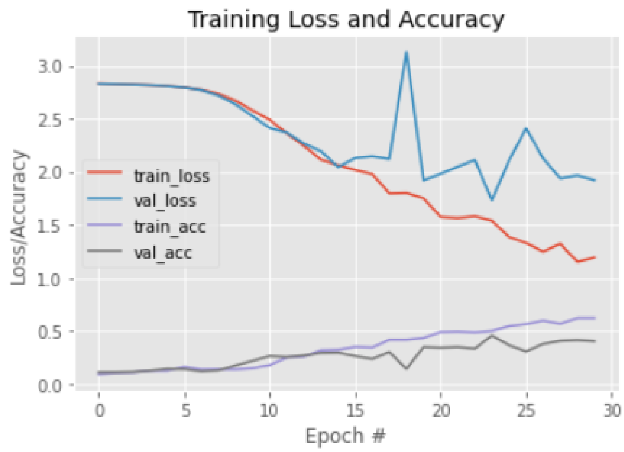

Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc

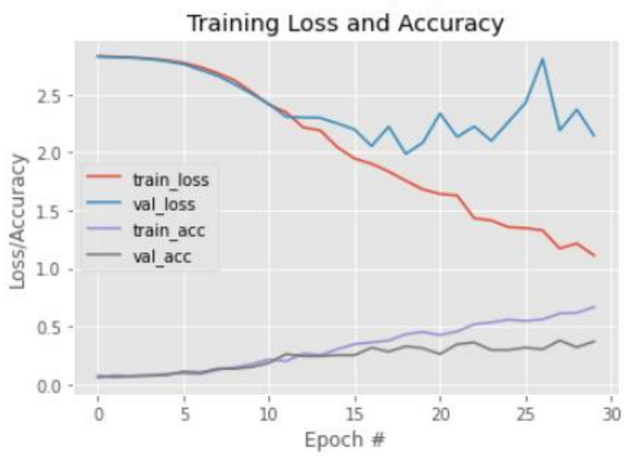

Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc

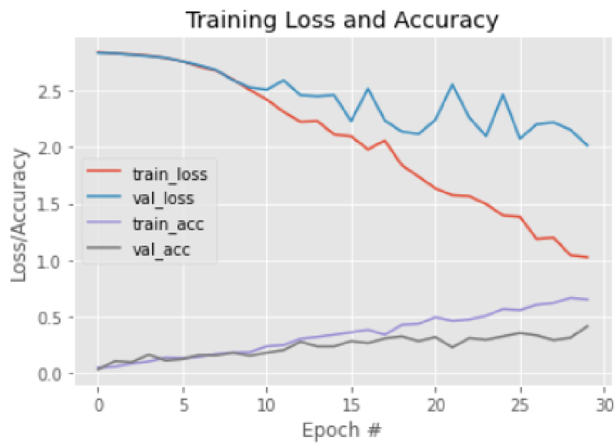

Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc

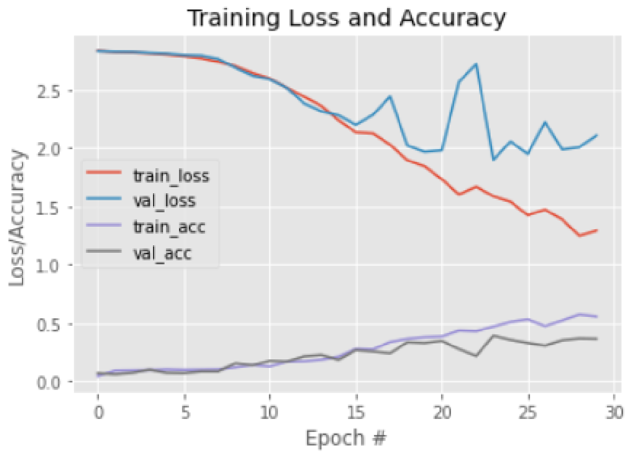

Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc

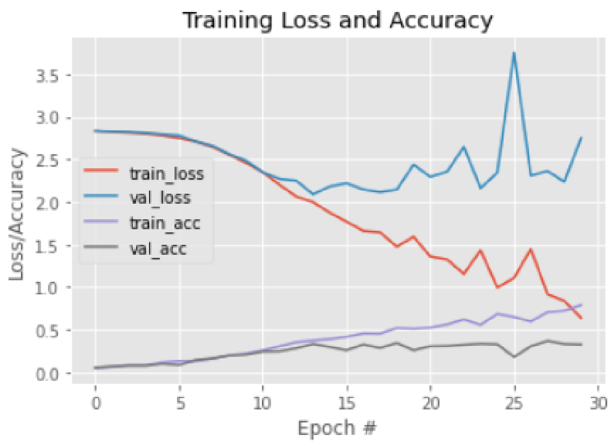

Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc


Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc


Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc


Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc





Model No	1	2	3	4	5	6	7	8	9	10	Ensemble
Best train accuracy (Epoch/percent)	30/62.94	30/67.45	30/61.57	30/66.86	30/64.51	30/61.96	30/66.67	29/66.67	29/57.25	30/78.82	NA
Best validation accuracy (Epoch/percent)	27/38.24	27/41.18	30/37.06	27/41.47	28/40.00	23/45.59	30/37.06	30/41.47	29/36.76	28/36.76	48.82%

The methodology you used for implementing the ensemble.

- Model Averaging on different model structure
  - In this following technique, I have used 4 different types of models and applied ensemble on the models
  - Given my validation accuracy of 54.70%
- Model Averaging on same model structure – 10 models with random initialization
  - In this following technique, I have used 10 same type of model, and every-time I have re-initialized the models and applied ensemble on the models
  - Given my validation accuracy of 51.57%
- Random Training Subset Ensemble
  - In this following technique, I have used 10 same type of model, and every-time I have re-initialized the models with sub-set of random data and applied ensemble on the models
  - Given my validation accuracy of 48.82%

Observations

- The best CNN in PART1 (i) given us 46.18% validation accuracy, and the least performing ensemble easily beats it.
- The baseline CNN give validation accuracy as 39.71% but the ensemble technique based on the baseline CNN give validation accuracies as 51.57% and 48.82%
- We can conclude that, ensemble technique is a powerful and good technique to get better predictions

The source of variability in your ensemble

- Different model structures
- Random initialization
- Random sub-set of data

why variability is an important factor when building an ensemble.

- If we use the same model, the ensemble will not give us any benefit. As we are essentially performing the same thing repeatedly

PART B: Transfer Learning

Pretrained VGG16 network

```
VGG_model = VGG16(weights="imagenet", include_top=False, input_shape=trainX.shape[1:])
VGG_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 128, 128, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 128, 128, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 128, 128, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 64, 64, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 64, 64, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 64, 64, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 32, 32, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 32, 32, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 32, 32, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 32, 32, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 16, 16, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 16, 16, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 16, 16, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 8, 8, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Feature extraction

```
features_train= VGG_model.predict(trainX).reshape(featuresTrain.shape[0], -1)
features_val= VGG_model.predict(valX).reshape(featuresVal.shape[0], -1)
```

RandomForest

```
model_RandomForest = RandomForestClassifier()
model_RandomForest.fit(features_train, trainY)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

```
results_RF = model_RandomForest.predict(features_val)
accuracy = accuracy_score(results_RF, valY)
print ("Accuracy (In percent): ", accuracy * 100)
correct = accuracy_score( results_RF,valY, normalize=False)
print("Imgaes found correctly: ",correct)
print("*"*50)
print("Confusion matrix :",confusion_matrix(valY, results_RF, labels=range(0,17)))
```

```
Accuracy (In percent):  79.41176470588235
Imgaes found correctly:  270
**************************************************
Confusion matrix : [[15  0  0  0  1  0  0  0  1  0  0  1  1  0  0  0  0]
 [ 0 16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  0 14  0  0  0  1  3  0  0  0  0  0  0  0  0  0]
 [ 5  0  0  8  0  1  0  0  0  0  1  0  2  0  0  2  1]
 [ 2  1  0  0 11  0  0  0  0  0  0  0  0  0  0  1  1]
 [ 0  0  0  1  0 15  0  0  0  0  0  0  0  0  0  1  1]
 [ 0  0  0  0  0  0 18  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0 19  0  0  0  0  0  1  0  0  0]
 [ 1  0  0  0  0  0  0  0 24  0  0  0  2  0  0  0  0]
 [ 0  0  0  0  1  0  0  0  0 20  1  0  0  0  0  0  0]
 [ 0  0  0  1  1  0  0  0  0  0 18  0  2  0  1  0  0]
 [ 1  0  0  1  0  0  0  0  0  0  0 19  0  0  0  0  2]
 [ 0  0  0  1  0  0  0  1  2  0  3  0 12  0  0  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  1  0 22  0  0  0]
 [ 0  1  0  0  1  0  0  0  0  0  0  0  0  0 17  0  0]
 [ 3  0  0  1  0  2  0  0  0  0  0  2  2  0  0  5  0]
 [ 1  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0 17]]
```

LogisticRegression

```
model= LogisticRegression()
model.fit(features_train, trainY)
model = model.predict(features_val)
accuracy = accuracy_score(model, valY)
print ("Accuracy (In percent): ", accuracy * 100)
correct = accuracy_score( model,valY, normalize=False)
print("Imgaes found correctly: ",correct)
print("*"*50)
print("Confusion matrix :",confusion_matrix(valY, model, labels=range(0,17)))
```

```
Accuracy (In percent):  87.94117647058823
Imgaes found correctly:  299
**************************************************
Confusion matrix : [[15  0  0  1  0  0  0  0  0  0  0  1  2  0  0  0  0]
 [ 0 16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  0 16  0  0  0  0  2  1  0  0  0  0  0  0  0  0]
 [ 1  1  0 14  1  1  0  0  1  0  0  0  0  0  0  1  0]
 [ 0  0  0  0 16  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0 16  0  0  0  0  0  0  0  0  0  1  0]
 [ 0  0  0  0  1  0 17  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0 20  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  0  0  0  0  0 25  0  0  0  1  0  0  0  0]
 [ 0  0  0  0  1  0  0  0  0 20  1  0  0  0  0  0  0]
 [ 1  0  0  0  1  0  0  0  0  0 21  0  0  0  0  0  0]
 [ 1  0  0  0  1  0  0  0  1  0  0 19  0  0  0  0  1]
 [ 1  0  0  1  1  0  0  0  0  0  1  0 15  0  1  0  0]
 [ 0  1  0  0  0  0  0  0  0  0  0  0  0 22  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0 19  0  0]
 [ 3  0  0  1  0  0  0  0  0  0  0  0  2  0  0  9  0]
 [ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0 19]]
```

LinearSVC

```
model= LinearSVC()
model.fit(features_train, trainY)
model = model.predict(features_val)
accuracy = accuracy_score(model, valY)
print ("Accuracy (In percent): ", accuracy * 100)
correct = accuracy_score( model,valY, normalize=False)
print("Imgaes found correctly: ",correct)
print("*"*50)
print("Confusion matrix :",confusion_matrix(valY, model, labels=range(0,17)))
```

```
Accuracy (In percent):  87.94117647058823
Imgaes found correctly:  299
**************************************************
Confusion matrix : [[15  0  0  1  0  0  0  0  0  0  0  1  2  0  0  0  0]
 [ 0 16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 1  0 16  0  0  0  0  2  1  0  0  0  0  0  0  0  0]
 [ 1  1  0 15  0  1  0  0  1  0  0  0  0  0  0  1  0]
 [ 0  0  0  0 16  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0 16  0  0  0  0  0  0  0  0  0  1  0]
 [ 0  0  0  0  1  0 17  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0 20  0  0  0  0  0  0  0  0  0]
 [ 0  1  1  0  0  0  0  0 25  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0  0  0 20  1  0  0  0  0  0  0]
 [ 1  0  0  0  1  0  0  0  0  0 21  0  0  0  0  0  0]
 [ 1  0  0  0  1  0  0  0  0  0  0 19  0  0  0  0  2]
 [ 2  0  0  0  1  0  0  0  0  0  0  0 16  0  1  0  0]
 [ 0  1  0  0  0  0  0  0  0  0  0  0  0 22  0  0  0]
 [ 0  0  0  0  1  0  0  0  0  0  0  0  0  0 18  0  0]
 [ 3  0  0  0  0  0  0  0  0  0  0  0  2  0  1  9  0]
 [ 0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0 18]]
```

Question (ii): Explore the application of fine-tuning as a method of transfer learning for the Flowers dataset.

- Step 1 & 2: Freezing all the layers & added a new fully connected layer

```
vgg_model = VGG16(weights="imagenet", include_top=False, input_shape=trainX.shape[1:])
vgg_model.summary()
vgg_model.trainable = False
```

```
model = Sequential()
model.add(vgg_model)
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(17, activation='softmax'))
```

- Step 3: Train the weights on the new FC layer.

```
epochs = 30
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=SGD(lr=0.01),
              metrics=['accuracy'])
checkpoint = tf.keras.callbacks.ModelCheckpoint("FineTune01bestVGG.h5", monitor="val_loss", mode="min", save_best_only=True, verbose=1)
history = model.fit(x=trainX, y=trainY,
                    batch_size=32,
                    epochs=epochs,
                    callbacks=[checkpoint],
                    validation_data=(valX, valY))
showGraph(histroy, epochs)
```

	CNN4
Overfitting epoch	22
Noise graphs	Slight noise
Best train accuracy (Epoch/percent)	30/95.10%
Best validation accuracy (Epoch/percent)	27/87.94%
Train accuracy at the overfitting epoch	88.92%
Validation accuracy at the overfitting epoch	85.95%

- Step 4: Unfreeze the trainable weights on some of the convolutional layers in the base network.

```
vgg_model.trainable = True
set_trainable = False
for layer in vgg_model.layers:
    if layer.name in ['block5_conv2']:
            set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

```
vgg_model.summary()
```

```
layers = []
for layer in vgg_model.layers:
    layers.append((layer, layer.name, layer.trainable))
print(pd.DataFrame(layers, columns=['Layer Type', 'Layer Name', 'Layer Trainable']))
```

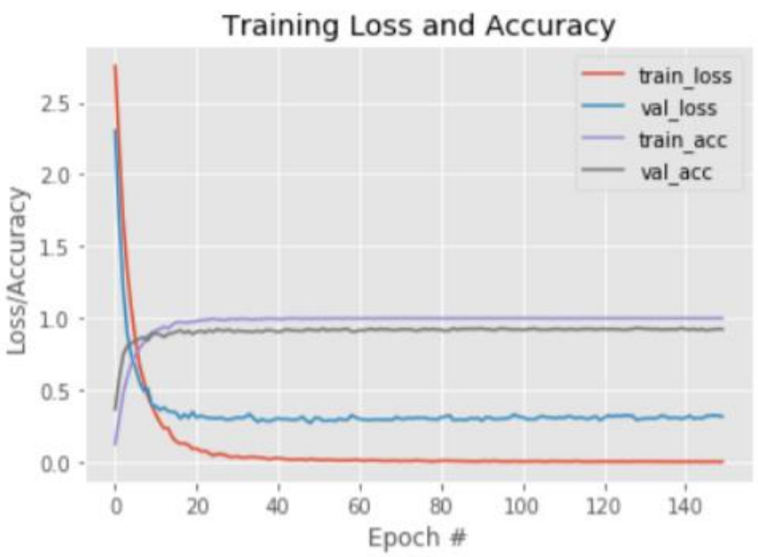
```
model = Sequential()
model.add(vgg_model)
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(17, activation='softmax'))
```

- Step 5: Train the network again using a very small training rate.

```
epochs = 150
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=SGD(lr=0.005),
              metrics=['accuracy'])
```

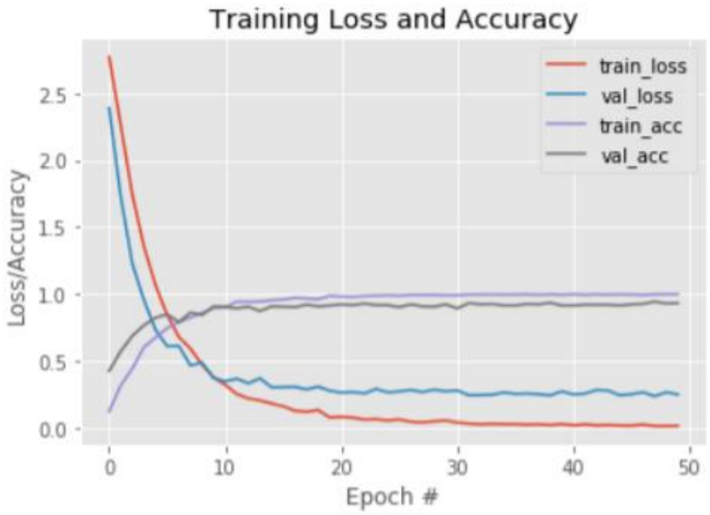
```
checkpoint = tf.keras.callbacks.ModelCheckpoint("postFineTunebestVGG.h5", monitor="val_loss", mode="min", save_best_only=True, verbose=1)
history = model.fit(x=trainX, y=trainY,
                              batch_size=32,
                              epochs=epochs,
                              callbacks=[checkpoint],
                              validation_data=(valX, valY))
showGraph(histroy, epochs)
```

```
showGraph(histroy, epochs)
```



	CNN4
Overfitting	No clear sign
Noise graphs	Slight noise
Best train accuracy (Epoch/percent)	48/100%
Best validation accuracy (Epoch/percent)	52/92.94%
Train accuracy at flatten (no validation loss improve)	51 epoch/99.80%
Validation accuracy at(no validation loss improve)	51 Epoch/92.94%

Re-running the experiment for 50 epochs



	CNN4
Overfitting	No clear sign
Noise graphs	Slight noise
Best train accuracy (Epoch/percent)	49/99.90%
Best validation accuracy (Epoch/percent)	47/94.47%

Observations

- Fine-tuning helped a lot with the performance of the model
- We got an amazing 94.47% of validation accuracy, which is much better than all the previous experiments we have did of the data.

Part C: capsule networks

CNN is a network of neurons that uses Convolutions to decide. CNN is a depiction of brain visual cortex working, to make decisions. CNN is majorly used for image recognition. The concept of CNN is derived by the amazing work by David H. Hubel[1] and Torsten Nils Wiesel[2] that confirmed that the brains visual cortex contains neurons that help the brain to recognize the signals from the eyes in the form of images. The work by David H. Hubel and Torsten Nils Wiesel was further used to developed "neocognitron" [3], artificial neural network which led the foundation for various type of neural networks including Convolutional neural network.

The foundation of CNN was kept and CNN shown great results when it comes to image classification. But the implementation of CNN in AI has it's problem. CNN in AI uses layers of make the network deeper and also provides us with great results when we choose a good CNN model structure with a good amount of data. But every layer in CNN focuses on specific sub-area or a portion of the image and train itself. This method is great, but there are 2 issues.

- CNN becomes sensitive to placement of the features on an image
- A special relationship of features is not captured properly (due to sub-sampling)

CNN becomes sensitive to placement of the features on an image

Say we train a CNN model for detecting a person in an image (Say with high number of images – all standing), the model will be able to predict a validation image with a person in it with high accuracy, but the model may fail to identify the same validation image when passed flipped 90 degrees or passed as upside down. This is because, the model is only trained on images with people all standing, and no flipped images. The solution to this issue could be data augmentation. But still, with data augmentation, we are simply training the model with different versions of data, the CNN model is still sensitive to the placement of features in the images.

The spatial relationship of features is not captured properly

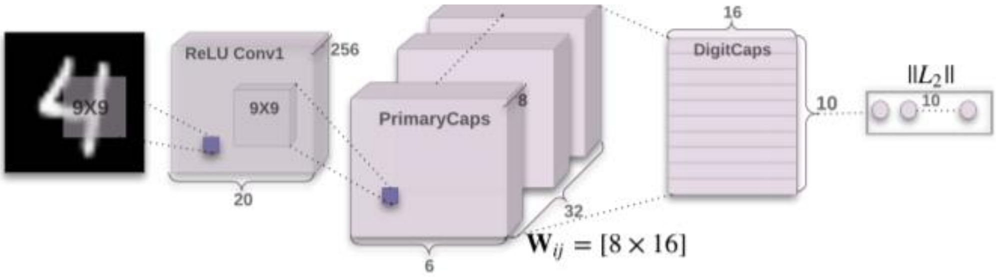
Now let's discuss a yet anther issue with CNN, for example, we have a trained model to identify a human face and used data-augmentation as well. Now the CNN is trained to identify features with eyes, nose, lips, facial structure, ears, hairs, etc. Now CNN doesn't capture spatial relationships due to sub-sampling (pooling), and we have trained the model with fabricated data using image augmentation, now the network is trained to identify specific features.

Now say we pass an image to this CNN, where we have eyes, nose, lips, ears, hairs in the image, but not on the face, but somewhere else. There is a high probability, that our CNN model will identify the image as a face.

Now the whole idea behind CNN is to depict the work of brain visual cortex working, but due to lack of Spatial information, a CNN model can be easily fooled. This same concept of fooling the CNN is very well explained in a recent paper "One-pixel attack for fooling deep neural networks" published [4] by Jiawei Su, Danilo Vasconcellos Vargas and Kouichi Sakurai.

Capsule Network:

Geoffrey Hinton, a well-known name suggested an alternate network architecture "Capsule Network"[5][6] to tackle the issues with pooling and suggested a way to also capture the directions of the features of the image. Capsule network (CapsNet) is a different type to a network where we use a different type of activation method "Squash" and instead of making the network deep, we use a group of neurons in a single layer. This group of neutrons is known as the capsules. Now in-order to train the model, we have to have a communication mechanism in a capsule. The communication is done by "dynamic routing". In-fact, "dynamic routing" and "Squash" are the hard and soul of the CapsNet



1. Architecture of CapsNet [6]

Activation function in CapsNet

Unlike the activation function in CNN, CapsNet doesn't use methods like ReLU, sigmoid, etc. It uses the squash method instead. The squash method returns Vj value. The value is been calculated using the Sj. Now Cij are the values from the capsules. There is no need for bias here. After discussing the variables, let's Discuss the important variable U the variable value contains the value and the direction values as well. When it comes to capsule networks the calculations are a vector-based not the linear type when compared to traditional neural networks. The squash activation function formula is given below along with a couple of more formulas that are used in the activation function of the capsule network.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

2. Squash formula [6]

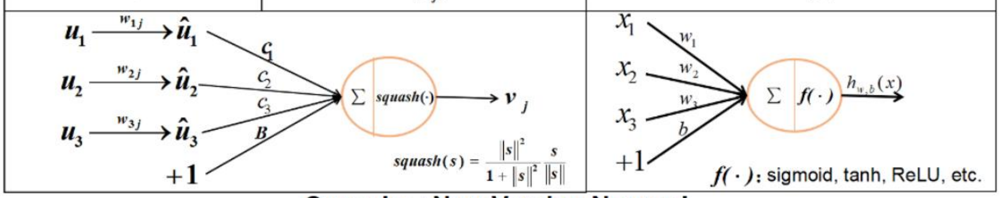
$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i} , \qquad \hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij} \mathbf{u}_i$$

3. Summation formula [6]

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

4. Capsule value formula [6]

		capsule	VS. traditional neuron
Input from low-level neurons/capsules		vector(u_i)	scalar(x_i)
Operations	Linear/Affine Transformation	$\hat{u}_{j\|i} = W_{ij} u_i + B_j$ (Eq. 2)	$a_{j\|i} = w_{ij} x_i + b_j$
	Weighting Summation	$s_j = \sum_i c_{ij} \hat{u}_{j\|i}$ (Eq. 2)	$z_j = \sum_{i=1}^{3} 1 \cdot a_{j\|i}$
	Non-linearity activation	$v_j = squash(s_j)$ (Eq. 1)	$h_{w,b}(x) = f(z_j)$
output		vector(v_j)	scalar(h)

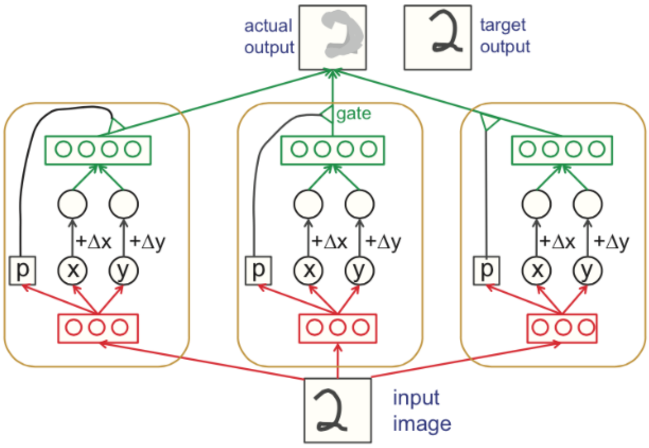


Capsule = New Version Neuron!
vector in, vector out VS. scalar in, scalar out

5. Capsule Vs Traditional neuron [7]

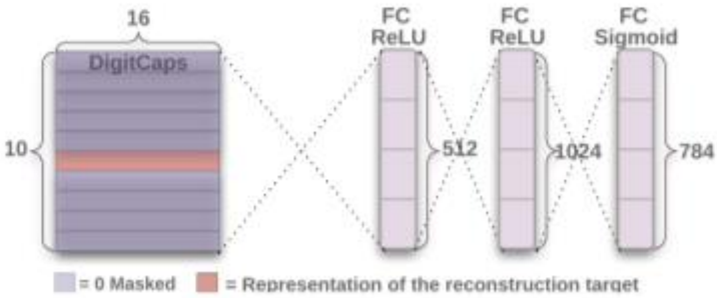
Dynamic routing

Dynamic routing output from a capsule vector to higher-level capsules j. Shouting decisions are made by changing the scaler weights.



6. Dynamic routing

Now once we get a cab soon it is then decoded using a usual CNN mechanism and we use the activation methods like ReLU Sigma it and then we get output out of it.



7. Decoders [6]

1. https://en.wikipedia.org/wiki/David_H._Hubel
2. https://en.wikipedia.org/wiki/Torsten_Wiesel
3. K. Fukushima, S. Miyake and T. Ito, "Neocognitron: A neural network model for a mechanism of visual pattern recognition," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 826-834, Sept.-Oct. 1983, doi: 10.1109/TSMC.1983.6313076.
4. https://arxiv.org/abs/1710.08864
5. https://link.springer.com/chapter/10.1007/978-3-642-21735-7_6
6. https://arxiv.org/abs/1710.09829
7. https://github.com/naturomics/CapsNet-Tensorflow/
8. http://helper.ipam.ucla.edu/publications/gss2012/gss2012_10754.pdf