Part A

Explores the application of convolutional networks, data augmentation and ensemble techniques.

Part A –(i)

Implement a baseline CNN, which contains just a single convolutional layer, single pooling layer, fully connected layer and SoftMax layer.

Baseline:

```
Architecture = Baseline
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 65536)             0
_________________________________________________________________
dense (Dense)                (None, 64)                4194368
_________________________________________________________________
dense_1 (Dense)              (None, 17)                1105
=================================================================
Total params: 4,195,921
Trainable params: 4,195,921
Non-trainable params: 0

Epoch 50/50
1020/1020 [==============================] - 4s 4ms/sample - loss: 0.0740 - acc: 0.9980 - val_loss: 1.7159 - val_acc: 0.5794
Total Time Elapsed 4.620654201507568  mins
```

Architecture 1:

```
Architecture 1
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 32768)             0
_________________________________________________________________
dense (Dense)                (None, 64)                2097216
_________________________________________________________________
dense_1 (Dense)              (None, 17)                1105
=================================================================
Total params: 2,103,409
Trainable params: 2,103,409
Non-trainable params: 0

Epoch 50/50
1020/1020 [==============================] - 7s 7ms/sample - loss: 0.1632 - acc: 0.9686 - val_loss: 2.1670 - val_acc: 0.4676
Total Time Elapsed 5.6428070346514385  mins
```

Architecture 2:

```
Architecture 2
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 128, 128, 16)      1216
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 64, 64, 32)        4640
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                1048640
_________________________________________________________________
dense_5 (Dense)              (None, 17)                1105
=================================================================
Total params: 1,074,097
Trainable params: 1,074,097
Non-trainable params: 0

    Epoch 49/50
1020/1020 [==============================] - 16s 16ms/sample - loss: 0.2757 - acc: 0.9373 - val_loss: 2.2405 - val_acc: 0.4971
Epoch 50/50
1020/1020 [==============================] - 16s 16ms/sample - loss: 0.3369 - acc: 0.9216 - val_loss: 2.0267 - val_acc: 0.5059
Total Time Elapsed 11.693238914012909  mins
```

Architecture 3:

```
Architecture = 3
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 128, 128, 16)      1216
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 64, 64, 32)        4640
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 96)        55392
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 8, 8, 96)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 6144)              0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                393280
_________________________________________________________________
dense_7 (Dense)              (None, 17)                1105
=================================================================
Total params: 474,129
Trainable params: 474,129
Non-trainable params: 0

Epoch 50/50
1020/1020 [==============================] - 27s 26ms/sample - loss: 0.6733 - acc: 0.8069 - val_loss: 2.2704 - val_acc: 0.4000
Total Time Elapsed 20.73042186896006  mins
```

Architecture 4:

```
Architecture 4
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 96)        55392
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 96)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 128)       110720
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 64)                524352
_________________________________________________________________
dense_1 (Dense)              (None, 17)                1105
=================================================================
Total params: 715,153
Trainable params: 715,153
Non-trainable params: 0

Epoch 50/50
1020/1020 [==============================] - 19s 18ms/sample - loss: 0.7300 - acc: 0.7578 - val_loss: 1.5882 - val_acc: 0.5294
Total Time Elapsed 21.837236114343007  mins
```

The Table below represents the overall performance of baseline and the other architectures.

Architecture	Train loss	Validation loss	Train Acc	Validation Acc	Overfitting?
Baseline	0.074	1.7159	0.998	0.5794	Major
Architecture 1	0.1632	2.167	0.9686	0.4676	Major
Architecture 2	0.3369	2.02	0.9216	0.5059	Major
Architecture 3	0.6733	2.2704	0.8069	0.4	Relatively Less
Architecture 4	0.73	1.5882	0.7578	0.5294	Relatively Less

My Comments:

- **Baseline:** This architecture contains only a CNN followed by a Max Pooling payer and finally a Fully Connected (FC) layer. This is a very basic architecture and hence a baseline. As expected, the model seems to perform better if we only consider validation accuracy number however, if we look at the graph it is evident that the model overfits with the Train Accuracy reaching almost 100% while the Validation Accuracy still looming around 58%. The model shows significant spikes on the performance charts which shows that the model is relatively unstable and the Gradient Descent is finding it hard to converge to the optimum.

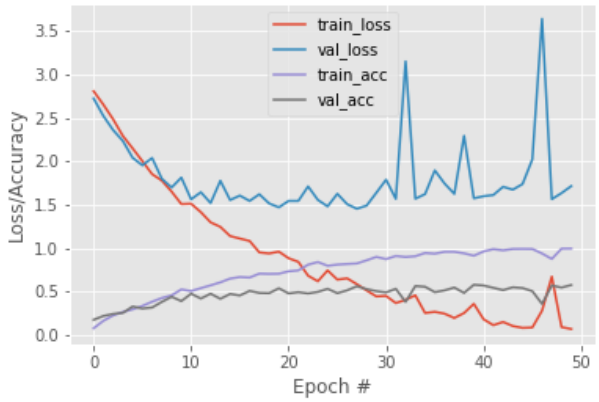
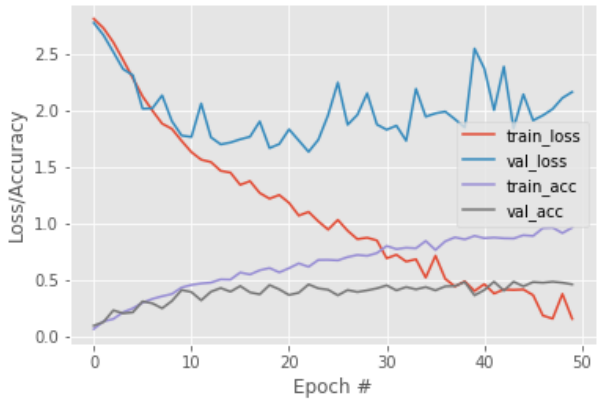
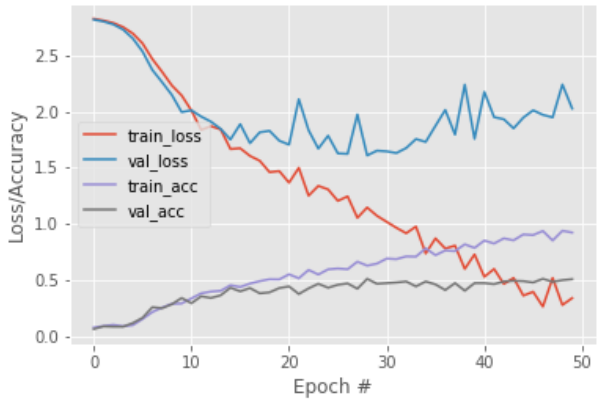
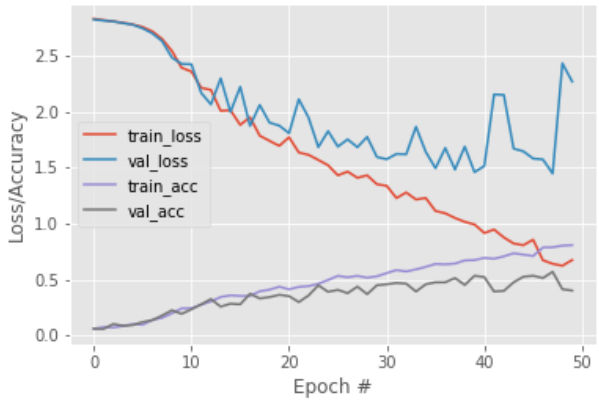
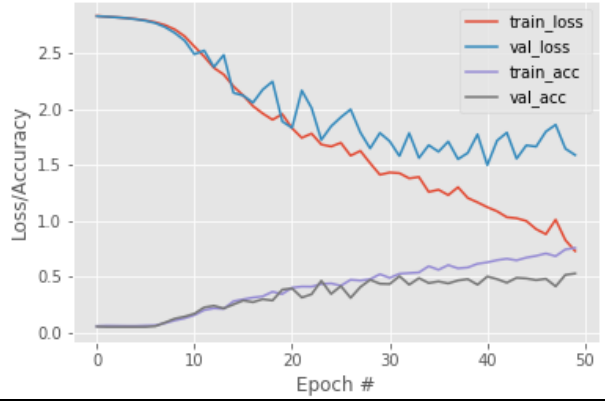
- **Architecture 1:** This architecture contains two CNN followed by a Max Pooling payer and finally a Fully Connected (FC) layer. This is an add on to the basic architecture. As expected, the model seems to perform better than the baseline. However, if we look at the graph it is evident that the model still overfits although not as much as the baseline. The spikes noticed in the baseline have subdued but not vanished. Clearly, an improvement from the baseline but not enough.

- **Architecture 2:** This architecture contains three CNN followed by a Max Pooling payer and finally a Fully Connected (FC) layer. This is an addition of an extra CNN and pooling layer to the Architecture 1. As expected, the model seems to perform better than the Architecture 1. However, if we look at the graph it is evident that the model still overfits although not as much as the Architecture 1. The spikes noticed have further subdued but not vanished. In the Architecture 1 we could notice overfitting from EPOCH 10 however in this architecture it happens after EPOCH 15. Again Clearly, an improvement from the predecessor but not enough.

- **Architecture 3:** This architecture contains four CNN followed by a Max Pooling payer and finally a Fully Connected (FC) layer. The model seems to perform better than the Architecture 2. However, if we look at the graph it is evident that the model still overfits although not as much as the Architecture 2. In the Architecture 2 we could notice overfitting from EPOCH which largely remain the same. However, in this model the rate at which the curves diverge reduces significantly. This shows, that the model is not as fast diverging from the optimum but it was. Again Clearly, an improvement from the predecessor but not enough.

- **Architecture 4:** In this model I could not notice much improvement from the previous model. However, what I noticed is that the Val loss remains plateaued at 1.5 in model 3 and model 4 but unlike model 3, model 4 looks very much stable and start to overfit after EPOCH 30.

- Below table shows the comparative performance of each of the architectures and the baseline:

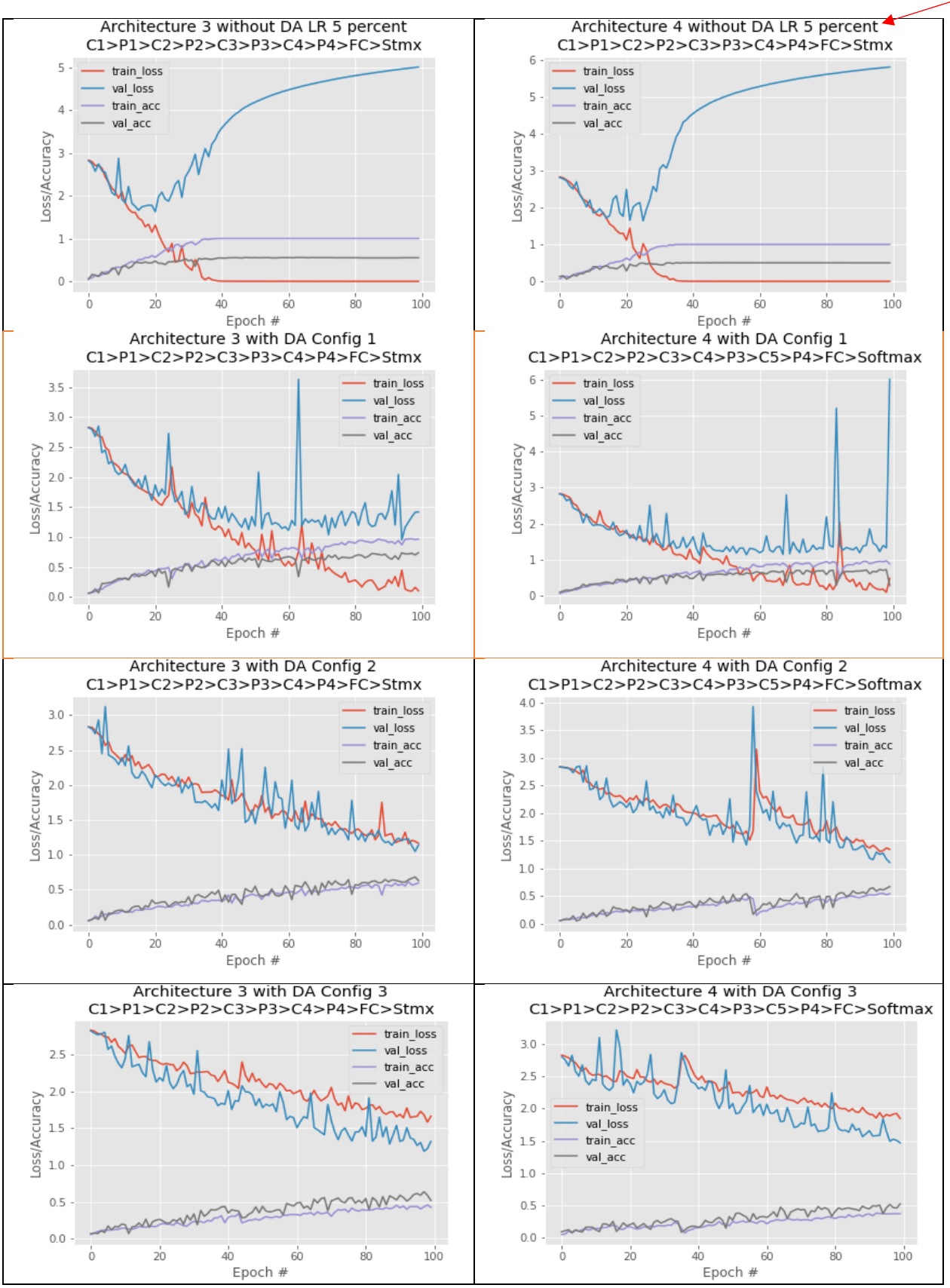
Baseline CNN1=>P1=>FC=>Softmax 	Architecture 1 CNN1=>P1=>CNN2=>P2=>FC=>Softmax 
Architecture 2 CNN1=>P1=>CNN2=>P2=>CNN3=>P3=>FC=>Softmax 	Architecture 3 =>P1=>CNN2=>P2=>CNN3=>P3=>CNN4=>P4=>FC=>S 
Architecture 4 1=>CNN2=>P2=>CNN3=>CNN4=>P3=>CNN5=>P4=>FC 	

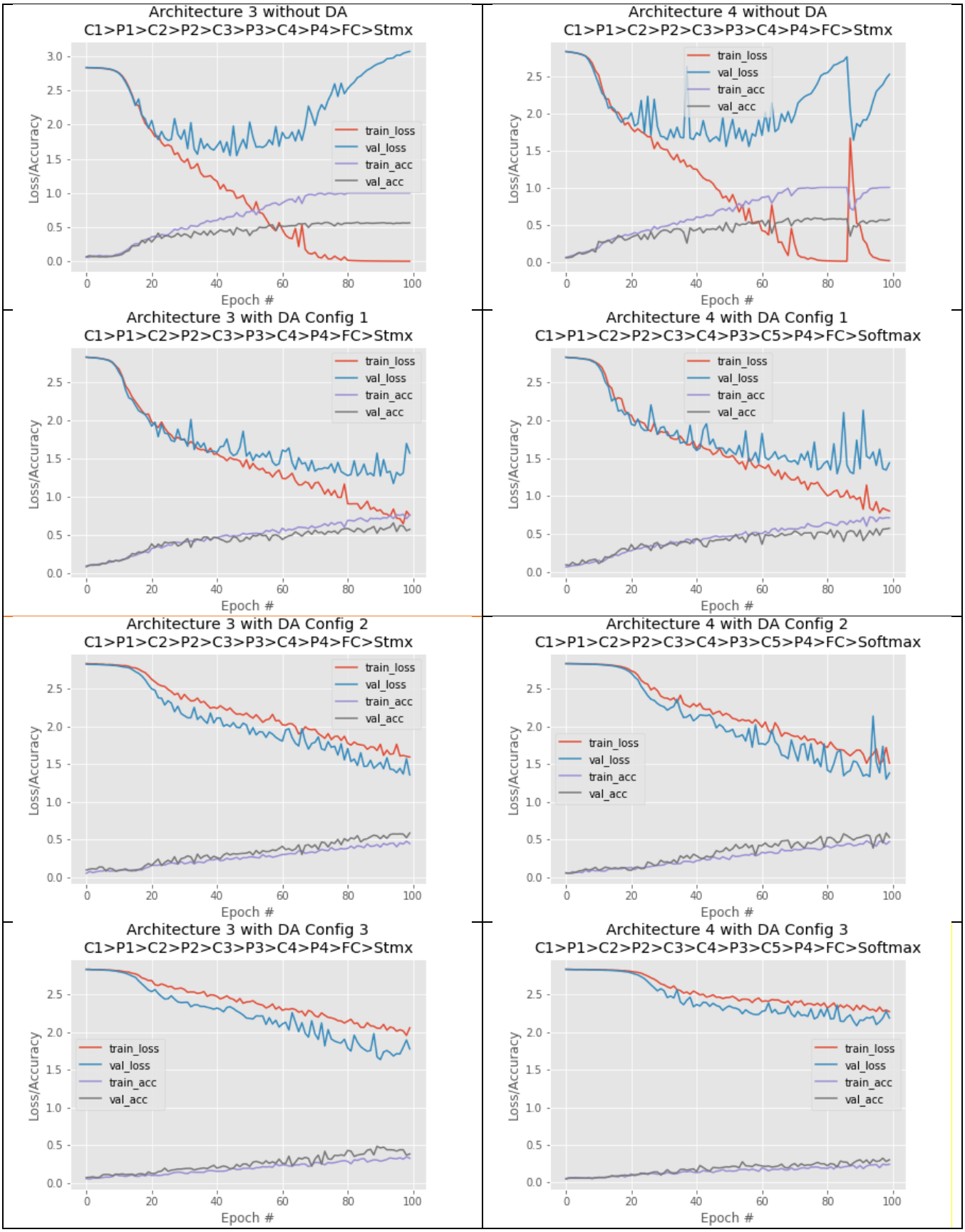
Investigate the implementation of data augmentation techniques for two deepest models.

- We shall investigate the impact of Data Augmentation (DA) on two of our deepest networks namely, Architecture 3 and Architecture 4.
- Following are the DA configurations we shall use.

Name	Rotation range	Width shift range	Height shift range	Shear range	Zoom range	Horizontal flip	Vertical flip	Fill mode
Config 1	10	0.1	0.1	0	0	False	False	Nearest
Config 2	45	0.33	0.33	0.33	0.33	True	False	Nearest
Config 3	90	0.5	0.5	0.5	1	True	True	Nearest

- Below are the performances of different network configurations.
  - Table 1 presents the performance at a learning rate of 0.05.
  - Table 2 presents the performance at a learning rate of 0.01.


Architecture 3 without DA LR 5 percent
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 without DA LR 5 percent
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 3 with DA Config 1
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 1
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
Architecture 3 with DA Config 2
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 2
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
Architecture 3 with DA Config 3
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 3
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #


Architecture 3 without DA
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 without DA
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 3 with DA Config 1
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 1
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
Architecture 3 with DA Config 2
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 2
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
Architecture 3 with DA Config 3
C1>P1>C2>P2>C3>P3>C4>P4>FC>Stmx
Architecture 4 with DA Config 3
C1>P1>C2>P2>C3>C4>P3>C5>P4>FC>Softmax
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #

My Comment:

What has been presented above:

- 3 different Data Augmentation Configurations (DA).

Name	Rotation range	Width shift range	Height shift range	Shear range	Zoom range	Horizontal flip	Vertical flip	Fill mode
Config 1	10	0.1	0.1	0	0	False	False	Nearest
Config 2	45	0.33	0.33	0.33	0.33	True	False	Nearest
Config 3	90	0.5	0.5	0.5	1	True	True	Nearest

- A numerical and graphical (presented above) comparison of the performance of Architecture 3 and Architecture 4, with no DA, Config 1, Config 2, Config3 at a Learning Rate of 0.05.

Name (100 Epochs) (Learning Rate:0.05)	Architecture 3 (Max Validation Acc)	Architecture 4 (Max Validation Acc)
No Augmentation	0.555	
Config 1	0.7382	0.7206(epoch 99)
Config 2	0.6294	0.6647
Config 3	0.5206	0.5235

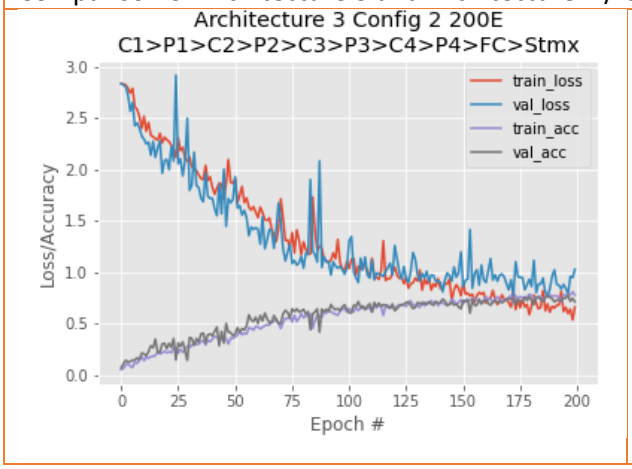
- A numerical and graphical (presented above) comparison of the performance Architecture 3 and Architecture 4, with no DA, Config 1, Config 2, Config3 at a Learning Rate of 0.01.

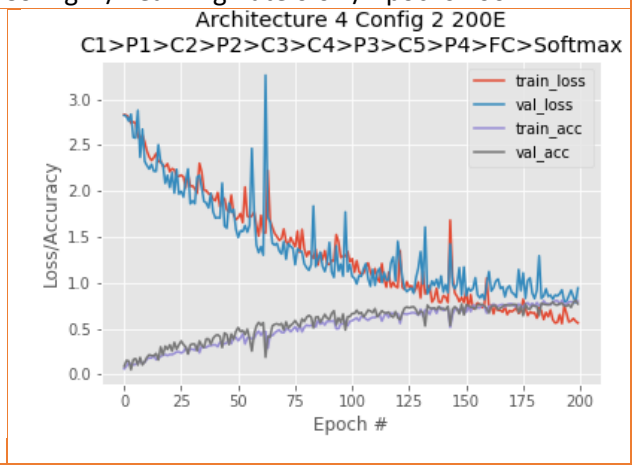
Name (100 Epochs) (Learning Rate:0.01)	Architecture 3 (Validation Accuracy)	Architecture 4 (Validation Accuracy)
No Augmentation	0.5647	0.5706
Config 1	0.6206 (epoch 97)	0.5765
Config 2	0.5853	0.5882 (epoch 99)
Config 3	0.4881 (epoch 90)	0.3853

- A Comparison of performance of Architecture 4 and Architecture 5, with the best Config (details in the table)

Architecture	Train loss	Validation loss	Train Acc	Validation Acc	Conclusion
Architecture 3	0.6606	1.02	0.7782	0.7118	Good
Architecture 4	0.562	0.9425	0.8	0.7706	Better

Comparison of Architecture 3 and Architecture 4 / Config 2 / Learning Rate 0.01 / Epochs 200





What did I learn?

- While performing evaluation on such networks, it is good idea to baseline a model and continue to go deeper unless to stop noticing improvements in the results.
- Having said this, this comes with a penalty of time to train and computation.
- **Data Augmentation Helps:** Although it might not always improve your model's accuracy but will most likely reduce overfitting.
- **Spend a good amount of time reviewing the data:** Before doing DA and coming up with the configurations, it is very important that you spend a good amount of time reviewing that data. Upon doing this I noticed that most of the image are well centered and a significant vertical and horizontal shift won't help. Also, vertical flip wasn't required as images are mostly symmetrical. Surprisingly, these observations helped and results are evident, Config 2 worked better.
- **The lesser learning rate the better, however mostly going below 1 % doesn't help much unless you are in the last mile of tuning the network.**

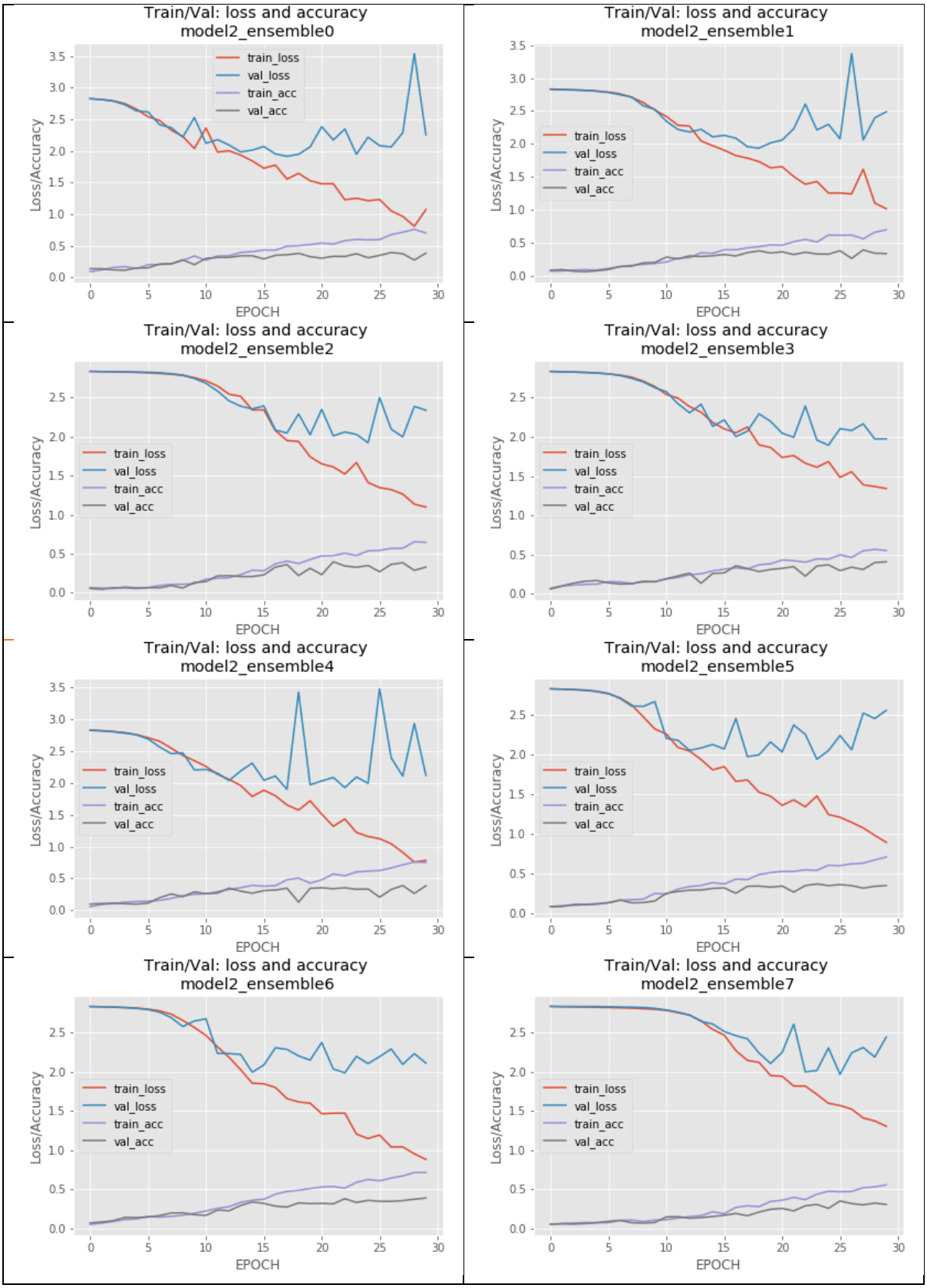
PART A (II) – Ensemble Learning

Model 1 - Variability in underling training data.

- This model was created with 8 base learners**, each trained on a different set of randomized training set and random weight initialization.**
- Hence, each configured with different weights.
- Combine output of each base learners and provide the average of the 8 base learners as the final output.
- Below are performances of each of the base learners.

```
****************************************
Ensemble accuracy Score:  0.4470588235294118
****************************************
```

- This was the overall performance of the Meta Learner.


Train/Val: loss and accuracy
model2_ensemble0
Train/Val: loss and accuracy
model2_ensemble1
Train/Val: loss and accuracy
model2_ensemble2
Train/Val: loss and accuracy
model2_ensemble3
Train/Val: loss and accuracy
model2_ensemble4
Train/Val: loss and accuracy
model2_ensemble5
Train/Val: loss and accuracy
model2_ensemble6
Train/Val: loss and accuracy
model2_ensemble7
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
EPOCH

Model 2 - Variability in the base learners.

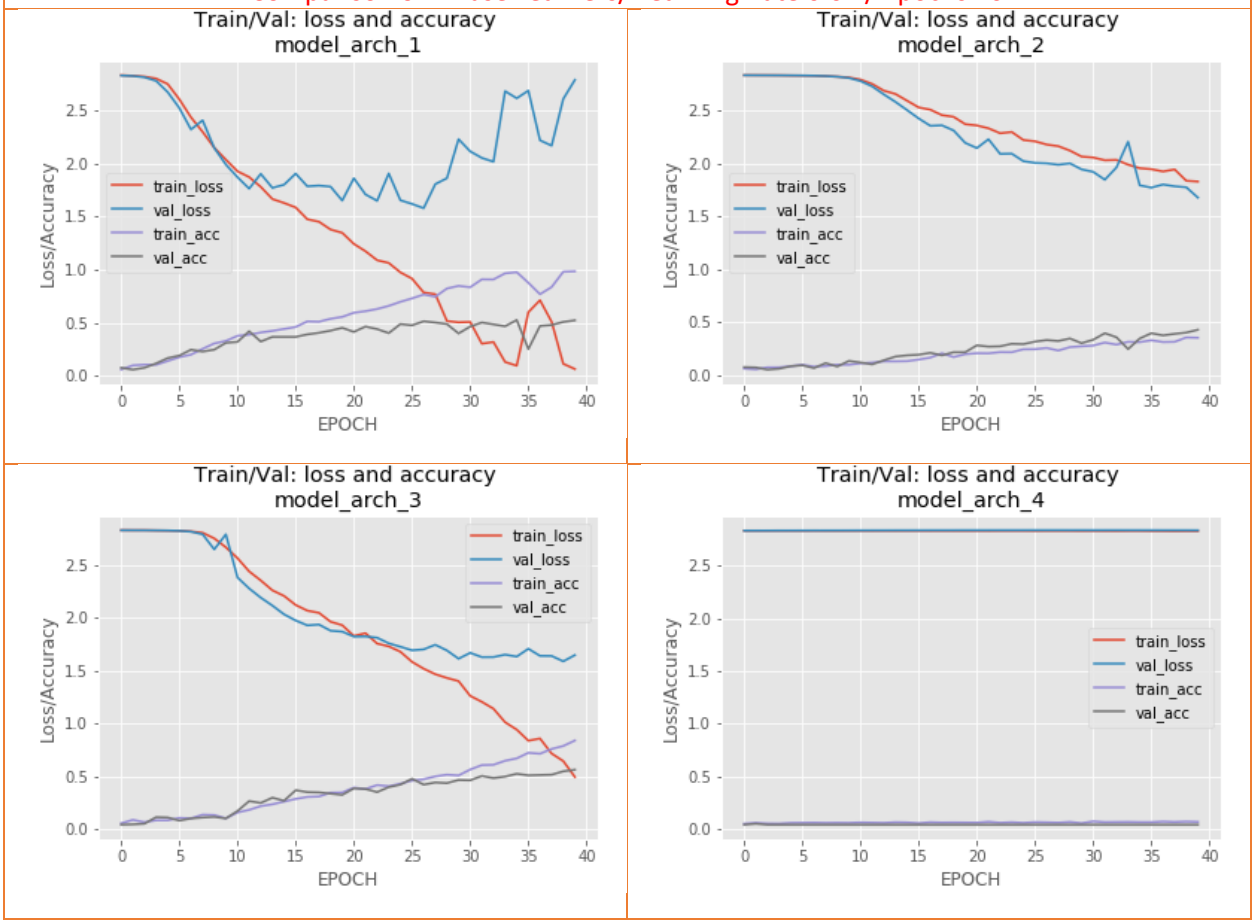
This model was created with **4 different base learners, each trained on the same set of training data.**

- Combine output of each base learners and provide the average of the 4 base learners as the final output.
- Below are performances of each of the base learners.
- This was the overall performance of the Meta Learner.

```
**************************************
Ensemble accuracy Score:  0.5794117647058824
**************************************
```

Architecture	Train loss	Validation loss	Train Acc	Validation Acc
Base Learner 1	0.0646	2.6099	0.9814	0.5235
Base Learner 2	1.8292	1.6776	0.3539	0.4294
Base Learner 3	0.4934	1.6492	0.8392	0.5618
Base Learner 4	2.283	2.8362	0.0706	0.0441

Comparison of 4 Base Learners/ Learning Rate 0.01 / Epochs 40



PART B (I)

Inception

```
Model: "inception_v3"
__________________________________________________________________________________________________
Total params: 21,802,784
Trainable params: 21,768,352
Non-trainable params: 34,432
```

Clubbing the pre-trained Inception V3 with Multiple Supervised Learning Algorithms:

KNeighborsClassifier

```
model_KNN = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=10, p=2,
           weights='uniform')

model_KNN.fit(featuresTrain, trainY)

results_KNN =model_KNN.predict(featuresVal)

print (accuracy_score(results_KNN, valY))

print("\n Number of correctly identified imgaes: ",accuracy_score( results_KNN,valY, normalize=False),"\n")
print("\n Confusion matrix : \n\n",confusion_matrix(valY, results_KNN, labels=range(0,17)))
```

```
Algorithm: InceptionV3 + RandomForest
Validation Accuracy : 0.8029411764705883
Number of correctly identified imgaes:  273
```

Logistic Regression

```
model_LR= LogisticRegression()
model_LR.fit(featuresTrain, trainY)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

```
Algorithm: InceptionV3 + Logistic Regressor
Validation Accuracy : 0.8735294117647059
Number of correctly identified imgaes:  297
```

Linear SVC

```
model_SVM= LinearSVC()
model_SVM.fit(featuresTrain, trainY)
```

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

```
Algorithm: InceptionV3 + Linear SVC
Validation Accuracy : 0.879429411
Number of correctly identified imgaes:  299
```

KNeighborsClassifier

```
model_KNN = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=10, p=2,
           weights='uniform')

model_KNN.fit(featuresTrain, trainY)

results_KNN =model_KNN.predict(featuresVal)

print (accuracy_score(results_KNN, valY))

print("\n Number of correctly identified imgaes: ",accuracy_score( results_KNN,valY, normalize=False),"\n")
print("\n Confusion matrix : \n\n",confusion_matrix(valY, results_KNN, labels=range(0,17)))
```

```
Algorithm: InceptionV3 + KNeighborsClassifier
Validation Accuracy : 0.6676470588235294
Number of correctly identified imgaes:  227
```

VGG16

Clubbing the pre-trained VGG16 with Multiple Supervised Learning Algorithms

```
Model: "vgg16"
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

```
Algorithm: vgg16 + KNeighborsClassifier
Validation Accuracy : 0.6705882352941176
Number of correctly identified imgaes:  228
```

```
Algorithm: vgg16 + RandomForest
Validation Accuracy : 0.8235294117647058
Number of correctly identified imgaes:  280
```

```
Algorithm: vgg16 + Logistic Regressor
Validation Accuracy : 0.8735294117647059
Number of correctly identified imgaes:  297
```

```
Algorithm: vgg16 + Linear SVC
Validation Accuracy : 0.8794117647058823
Number of correctly identified imgaes:  299
```

ResNet50V2

Clubbing the pre-trained ResNet50V2 with Multiple Supervised Learning Algorithms

```
Model: "resnet50v2"
Total params: 23,564,800
Trainable params: 23,519,360
Non-trainable params: 45,440
```

```
Algorithm: resnet50v2 + KNeighborsClassifier
Validation Accuracy : 0.7029854451225412
Number of correctly identified imgaes:  235
```

```
Algorithm: resnet50v2 + RandomForest
Validation Accuracy : 0.8294184221515500
Number of correctly identified imgaes:  282
```

```
Algorithm: resnet50v2 + LogisticRegression
Validation Accuracy : 0.9029294117645487
Number of correctly identified imgaes:  307
```

```
Algorithm: resnet50v2 + LinearSVC
Validation Accuracy : 0.8767402254100128
Number of correctly identified imgaes:  298
```

Performance of Different Pre trained Networks with various Supervised Algorithms.

Pre-Trained Network/ Supervised Learning Algo	KNN	Random Forest	Logistic Regressor	Linear SVC	Conclusion
InceptionV3	0.6676	0.8029	0.8735	0.8794	InceptionV3+Linear SVC
VGG-16	0.6705	0.8235	0.8735	0.8794	VGG-16 + Linear SVC
ResNet50V2	0.7029	0.82941	0.9029	0.87674	ResNet50V2 + Logistic

My Remarks (Rationale for using these algorithms):

- I have used **Inception V3, VGG16, ResNet50V2** as three variants of the pretrained networks.
- Inception has been seen to perform better than the VGG16 and ResNet50V2 in terms of accuracy in the ImageNet Competition and also has significantly less trainable parameters which leads to shorter convolution time. **This is something I wanted to verify on this dataset.**
- I wanted to see if there are any clusters in the data of which we can take an advantage of, **so I decided to KNN.** Despite using number of combinations, the algorithm does not seem to perform so well.
- I checked if an ensemble algorithm would do well, I tried Random Forest and Gradient Boosted Trees (GBT**). Random Forest seemed to perform relatively better than KNN.** I could not produce results of GBT as it takes too long to train.
- **I wanted to check if a single entity than an ensemble would work better. I asked read some researches where Linear SVM, Logistic and Decision Tree have shown to perform better in such settings**. As expected, Linear SVC and Logistic regressor performed equally good with both Inception and VGG.
- However, Logistic Regressor performed the best by classifying 307/340 images correctly.

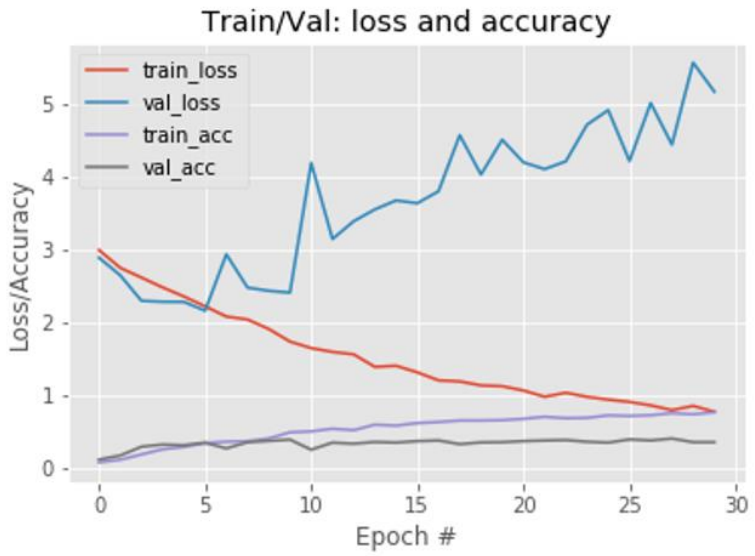
Fine Tuning the Model

Inception

With all Layers Freezed

```
INception V3 Freezed

Epoch 29/30
1020/1020 [==============================] - 69s 67ms/step - loss: 0.8559 - accuracy: 0.7412 - val_loss: 5.5588 - val_accuracy: 0.3588
Epoch 30/30
1020/1020 [==============================] - 69s 68ms/step - loss: 0.7753 - accuracy: 0.7676 - val_loss: 5.1603 - val_accuracy: 0.3588
```



Freezed Inception V3

- The results of Inception over this Data Set do not seem to be encouraging enough to proceed with fine tuning. Hence, we shall not continue with this network.

VGG16

Fine Tuning VGG 16

Performance of Freezed VGG16

```
VGG16 freezed 512/256
Epoch 30/30
1020/1020 [==============================] - 102s 100ms/step - loss: 0.2812 - accuracy: 0.9245 - val_loss: 0.4187 - val_accuracy: 0.8735
```

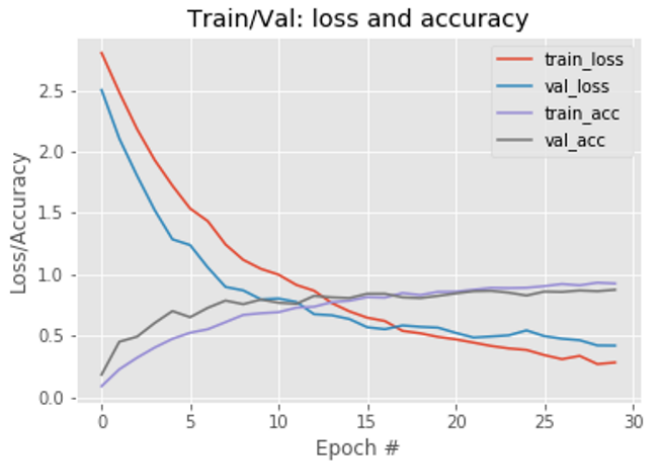
VGG 16 Unfreezed from Block 5

0	<keras.engine.input_layer.InputLayer object at...	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0...	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0...	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0...	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0...	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0...	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0...	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0...	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0...	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0...	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0...	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0...	block4_conv1	False
12	<keras.layers.convolutional.Conv2D object at 0...	block4_conv2	False
13	<keras.layers.convolutional.Conv2D object at 0...	block4_conv3	False
14	<keras.layers.pooling.MaxPooling2D object at 0...	block4_pool	False
15	<keras.layers.convolutional.Conv2D object at 0...	block5_conv1	True
16	<keras.layers.convolutional.Conv2D object at 0...	block5_conv2	True
17	<keras.layers.convolutional.Conv2D object at 0...	block5_conv3	True
18	<keras.layers.pooling.MaxPooling2D object at 0...	block5_pool	True

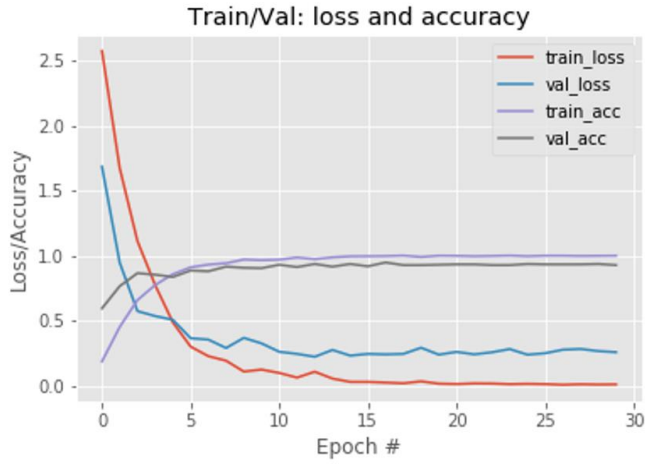
VGG 16 Unfreezed from Block 4

0	<keras.engine.input_layer.InputLayer object at...	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0...	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0...	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0...	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0...	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0...	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0...	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0...	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0...	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0...	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0...	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0...	block4_conv1	True
12	<keras.layers.convolutional.Conv2D object at 0...	block4_conv2	True
13	<keras.layers.convolutional.Conv2D object at 0...	block4_conv3	True
14	<keras.layers.pooling.MaxPooling2D object at 0...	block4_pool	True
15	<keras.layers.convolutional.Conv2D object at 0...	block5_conv1	True
16	<keras.layers.convolutional.Conv2D object at 0...	block5_conv2	True
17	<keras.layers.convolutional.Conv2D object at 0...	block5_conv3	True
18	<keras.layers.pooling.MaxPooling2D object at 0...	block5_pool	True

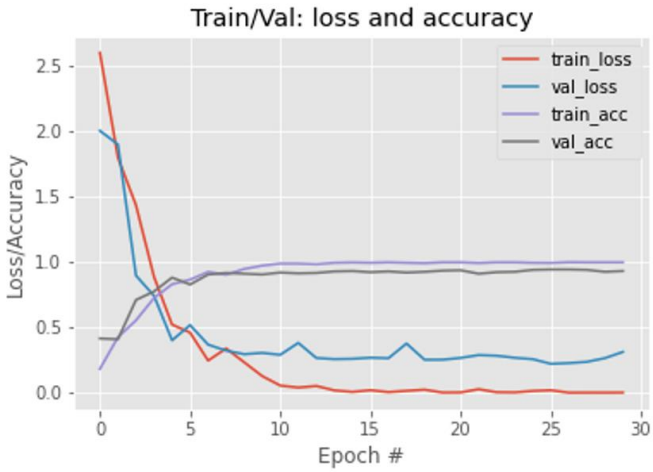
Performance of VGG 16 Network



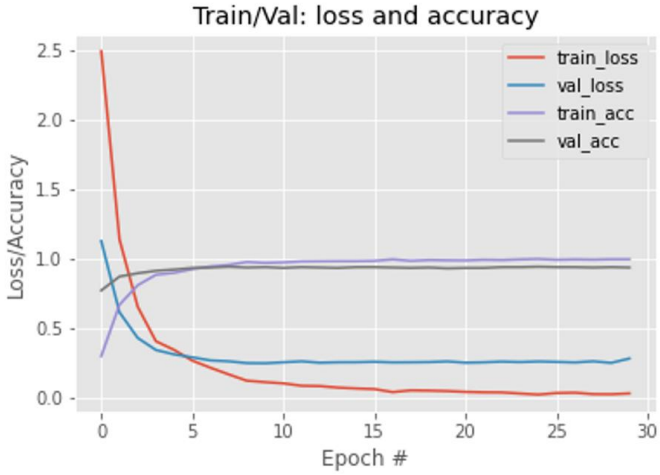
Freezed VGG16



VGG16: Block 5 Unfreezed



VGG16: Block 4 Unfreezed



VGG16: Block 4 Unfreezed LR:0.001

Performance of VGG 16 over configurations:

Architecture VGG-16 Epochs:30	Train loss	Validation loss	Train Acc	Validation Acc
All Freezed	0.2812	0.4187	0.9245	0.8735
Unfreezed Block 5 LR:0.01	0.0083	0.2656	0.9980	0.9353
Unfreezed Block 4 LR:0.01	0.0048	0.2394	0.999	0.9412
Unfreezed Block 4 LR:0.001	0.0208	0.2591	0.9971	0.9412

My Remarks:

- Inception Network does seem to perform very well with this dataset.
- With VGG, the performance continuously seems to improve as we go on to defreeze deeper layers and allow to network to train over the dataset.
- **Validation Loss: if** we observe the graphs carefully, we can notice that as we go deeper the validation loss starts from a lower level. For example, in the first one it starts from 2.5 however in the last it starts from almost 1. This is primarily because we allow a greater part of the network to adjust weights according to the dataset.
- **Overfitting:** does not seem to be a major issue in any of the networks however it continues to improve further as we go deeper in tuning the network.
- **Learning Rate:** if we carefully observe, we can notice minor bumps in the defreeze Block 4 LR 0.01 graph. These bumps appear to be smoothened in the graph with LR:0.001. This happens because it helps the optimizer to descend slower however in our case it does not come with a clear advantage given the amount of extra time and resources it takes to learn.
- **Validation Accuracy:** the accuracy continues to improve as we explore the deeper networks. We have already obtained a very high level of accuracy with block 5, which continues to improve by 0.6% with block 4. Accuracy is always a trade off and it is always a judgement call if to spend more resources for a marginal improvement beyond a certain point.
- **Optimizers:** SGD by far performs a lot better than Adam and Rmsprop.
- **Final Remark: with fine tuning we could manage to improve the performance (validation accuracy) from approx. 87% to 94%.**

PART C

Capsule Networks

The background:

Computer Vision applications have been widely discussed since decades but lost their steam in the early 90's, primarily due to two reasons,

1. **Insufficient Data to learn from** - As there was not enough digitalization like CCTVs and mobile phones.
2. **Insufficient processing power**: processors were not power enough to handle computation requiring millions of parameters to train.

The Resurgence:

With the **advancement in chip fabrication technology** it became possible to fabricate millions of transistors per cm^2 which led to exponential increase in processing power. As nanotechnology advanced, the processors finally became powerful enough to handle large computation applications such as computer vision. Nanotechnology and VLSI also paved a way for a digital revolution which led to a exponential rise in use of high end mobile phones and CCTVs.

While the stage was set, in 2012, Yann LeCun published a paper which suggested the use of Deep Convolution Neural Networks for Image Recognition which was crucial for Computer Vision applications. This paper was the pioneer work of Yann LeCun and team which we today know as **Convolutional Neural Networks (CNN)**. This paper broke all previous records of image processing algorithms and soon became a sensation.

Since then Deep Learning has been applied in a range of applications like medical, military, and space explorations and has most often if not always delivered far better results.

Researchers have been stacking layers over layer and have come up with a host of architectures for image recognition which has also led to Image Recognition competition like **ImageNet** where researchers around the world compete to deliver the best image recognition accuracy over a specific dataset.

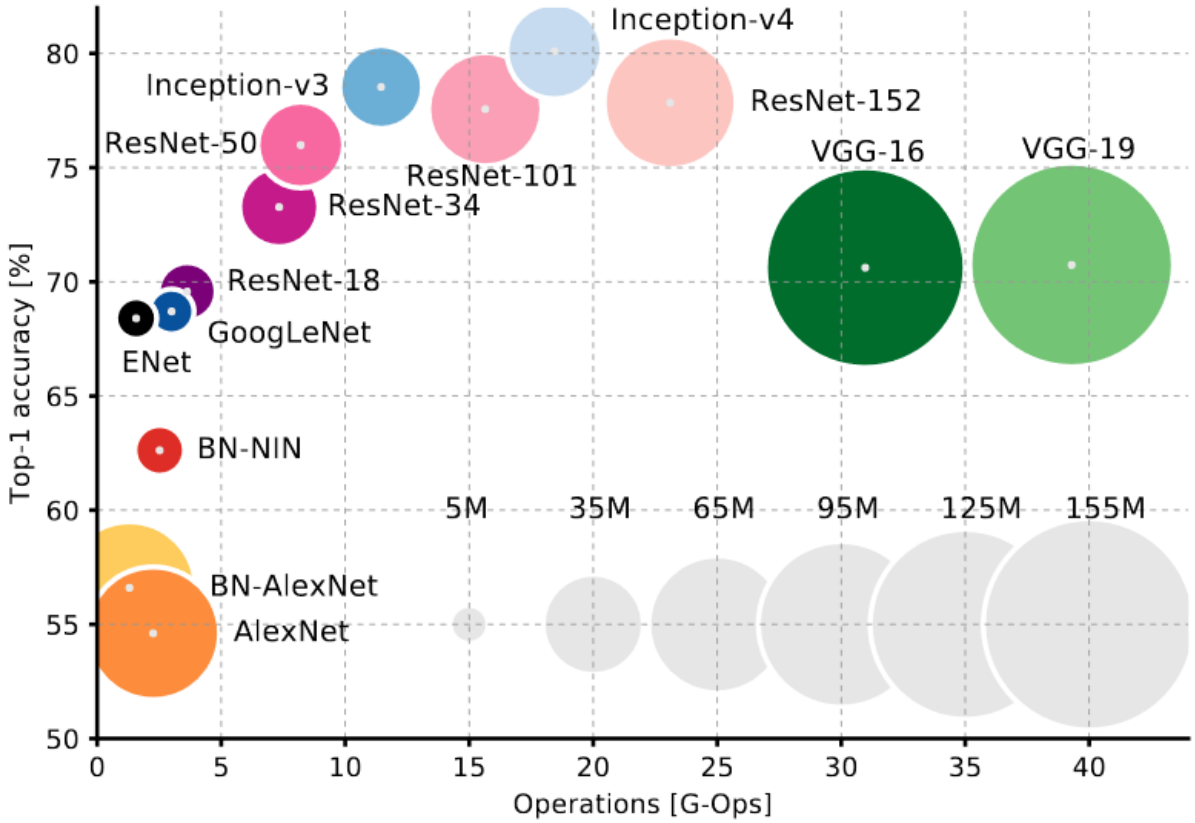


Figure 1: Compares the performance of the different Deep Neural Networks submitted to the Image Net Challenge over the last 4 years. The chart compares the accuracy of the network's v/s the number of learnable parameters each forward pass which may also be considered synonymous to the amount of time it takes to train the network. Source

What's the problem? Everything seems to be working pretty well!!

Problems with Deep Learning (CNN):

The main building block of a CNN is the convolution operation which in essence does is to detect features in an image. And then based on these key features the Neural Network is trained to update its weights in a way that those neurons get activated once certain features are detected.

Although this seems to be a decent approach but it has a major flaw. The CNN would still detect the object in an image even if the features do not share spatial relationship. This can be seen in the figure 3 below.

1. No spatial relationship retained.
   In a CNN the lower level features combine to form a higher-level feature which is a weighted sum of the lower level features. So, all it takes for a CNN to come up with a higher-level feature is for the lower level features to come together but with all this the

spatial relationship between them is not taken into account. This is a core design flaw in the CNN as they also rely on the scalar values.

The major challenge of a CNN that led to introduction of CapsNet is the inability of CNN to recognize pose, texture and deformation. (Sabour et al.,2017)

2. Needs a lot of data to train.

This comes as a consequence of the flaw that we discussed in point 1. As there is no spatial relationship maintained, the model will need to be trained in all the versions of the image possible. This becomes difficult when you have a limited data to train a model which can then be fixed to a certain extend by Data Augmentation.

Figure 4: Data Augmentation done right.

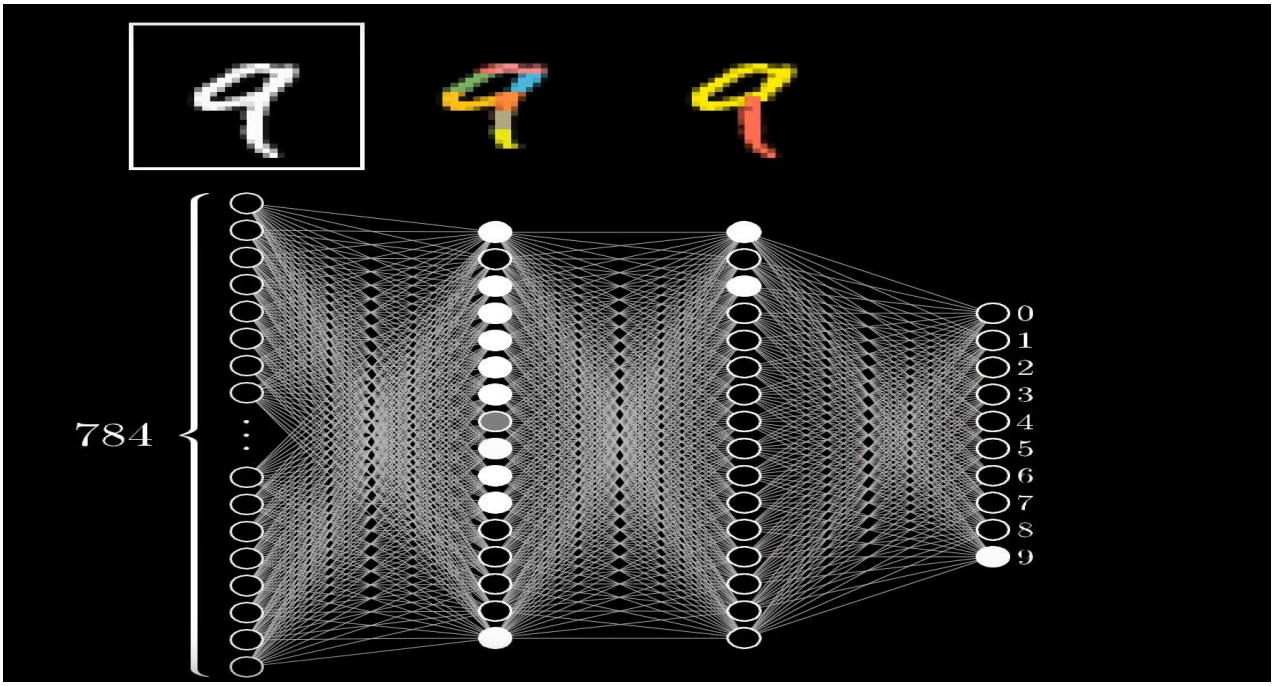


Figure 2: Shows how a neural network relies on certain neurons that are trained on certain features in an image which in turn ignite the subsequent neurons trained on broader features who in turn together help to classify the final image. source

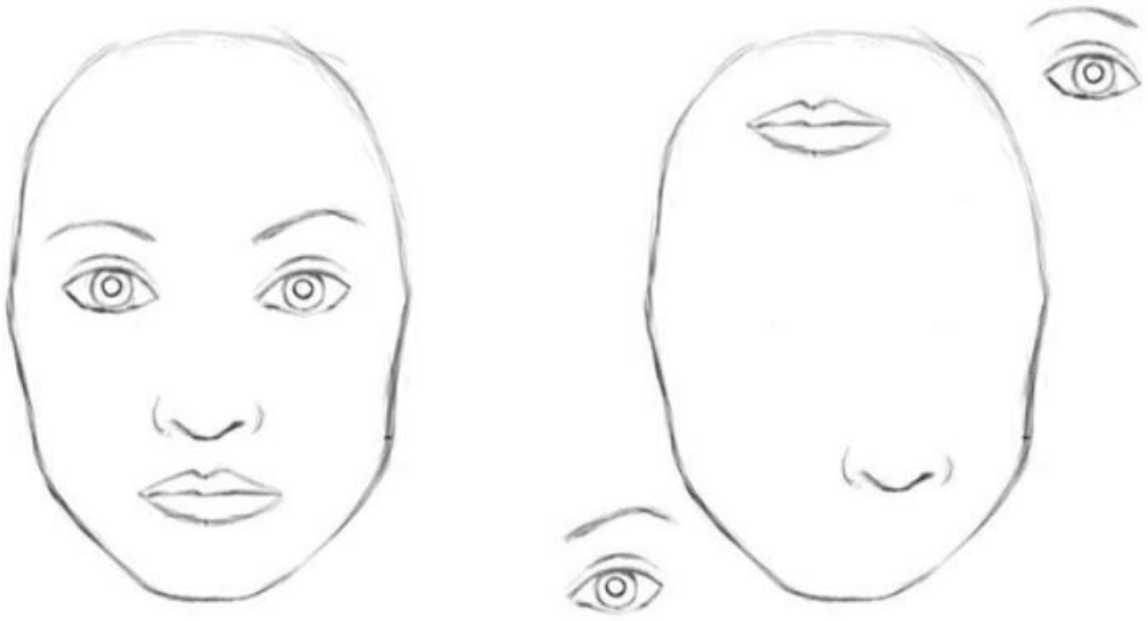


Figure 3: shows an example where a CNN would fail. source

How does Capsules alleviate this flaw?

Hinton argues that in order to correctly classify image it is important to retail the spatial relationship among the features as opposed to relying on just the presence of those features as in CNN. Hinton proposed **Capsule** as a new building blocks of deep neural networks which would help establish these spatial relationships among the features. It is imperative that once the models learn the spatial relationships among the features, it would not be very difficult for a model to infer that the given image is not a new one but just a transformation of an image it has already learned upon. **Hence, only a single image would give the network an idea of how this image would look like in different poses.** This is what Hinton calls "Knowledge" in his lecture on Capsule Networks.

The key to Hinton's innovation is the use of Vectors over Scalers in Image Recognition.(very important)

What are Capsules?

Capsules encapsulates all important information about the state of the feature they are detecting in a Vector Form.

Max Pooling:

Hinton in his lecture on YouTube says, the pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster.

He further goes on to explain that Max Pooling leads to information loss as only a so-called important feature of the image part is taken and rest all is discarded which leads to invariance. Invariance means that by changing the input a little that output remains the same. Which also means that even if the object in the image will shift then model will still detect the object which is not desirable.

However, Equivariance, which is a desirable property of a Capsule Network, will also consider the spatial location of the object and hence will not just consider the presence of nose in an image but also consider its overall location and size in proportion to the high-level feature.

There are three different implementations of Capsule.

1. Transforming Auto Encoders (Hinton et al.,2011)
2. Vector Capsules Based on Dynamic Routing (Sabour et a.,2017)
3. Matrix Capsules Based on expectation-maximization (Hinton et al., 2018)

How does a Capsule work?

Essentially a Capsule performs almost similar operation of features matrix transformation as in a normal neural network but the difference if all of this is **done in terms of Vectors rather than Scalers.**

Moreover, the **capsule performs one extra step** which can be seen as an activation function for a vector. In technical terms this is called as affline Transformation also called as *Squash*.

Below are the operations:

1. matrix multiplication of input vectors
2. scalar weighting of input vectors
3. sum of weighted input vectors
4. vector-to-vector nonlinearity (Squash)

1. **matrix multiplication of input vectors:** The input(U) to these capsules come from other low-level capsules that determine the low-level features. W is a matrix that encodes the spatial relationship between the low-level features and the current high-level feature being determined. For example, W1 might encode the relationship of nose to the overall face and W2 might encode the spatial relationship of the mouth or eyes with the overall face.

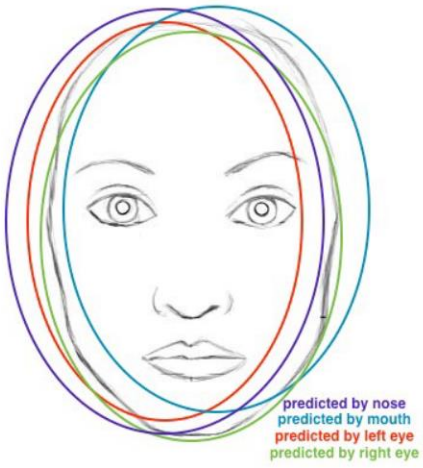


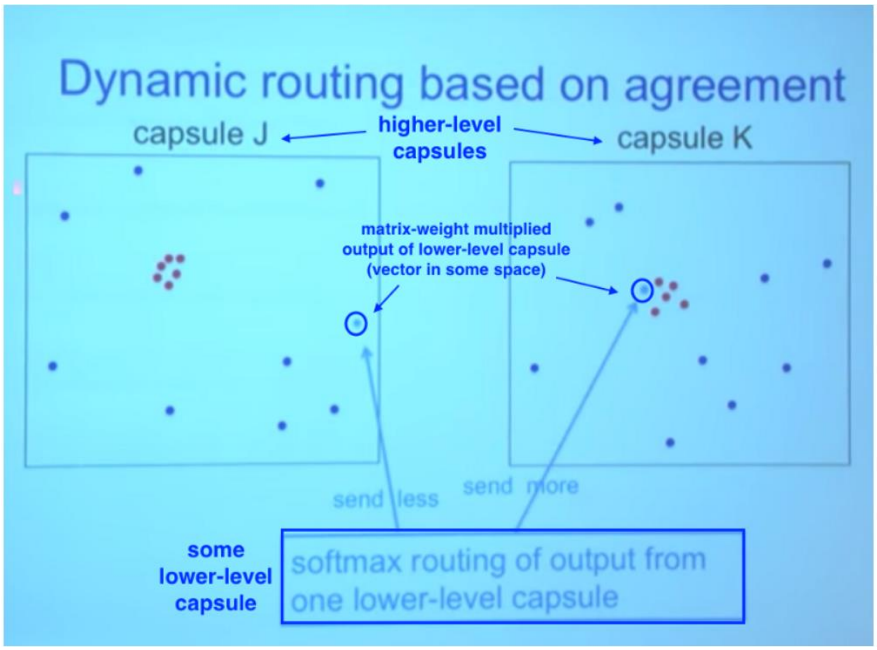
Figure: Prediction of face feature by different capsules representing low level features and their spatial relationship. (source)

Capsule vs. Traditional Neuron			
Input from low-level capsule/neuron		vector($\mathbf{u}_i$)	scalar(x_i)
Operation	Affine Transform	$\hat{\mathbf{u}}_{j\|i} = \mathbf{W}_{ij}\mathbf{u}_i$	–
	Weighting	$\mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j\|i}$	$a_j = \sum_i w_i x_i + b$
	Sum		
	Nonlinear Activation	$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1+\|\mathbf{s}_j\|^2}\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$	$h_j = f(a_j)$
Output		vector($\mathbf{v}_j$)	scalar(h_j)

Figure: Shows the operational difference between a Capsule Network and Neural Networks.

2. scalar weighting of input vectors:
   Like neural networks, capsule also weight the input feature vectors in terms of the weights. However, since Capsules use vector representations weighting is done using a novel approach called as Dynamic Routing (Routing by Agreement).

   Dynamic Routing forms the most popular part of this algorithm. Dynamic routing is a building block employed between layers. **This algorithm replaces pooling and scalar outputs in CNN with a vector output.** The magnitude of this output represents the likelihood that the feature being represented by the capsule is present in the image and the direction represents the orientation of the high-level feature (face in our case) as perceived by the capsule. As a result, the capsule network will not recognize an image with nose above the face line as a face. (Sabour et al., 2017)



Dynamic Routing is an algorithm which decides to which high level capsule should the low-level capsule output (Ci) go to. This is done by adjusting the weight such that the Ci goes to either Capsule K or J. For the blue circled capsule output will be multiplied in such a manner that it will represent its position on the spatial place as a vector specifying the orientation, size, and confidence. This location of the output (I) may end up the output being a part of a cluster of output giving similar results. The *I* might be different in J plane and K plane as each capsule has its own weights.

3. Sum of Weighted Input Vectors

Finally, the capsule with the maximum overall representations (bigger cluster) will end up as a final result. This is done using the sum of weighted input vectors.

4. "Squash": Novel Vector-to-Vector Nonlinearity

Squash is an activation function which converts the length of the output vectors into a length which is no more than 1.

References:

Hinton on Auto Encoders 2011

Hinton Matrix capsules with EM routing 2018

Sabour on Vector Capsules Based on Dynamic Routing 2017:

Hinton's' lecture at MIT

3Blue1Brown on YouTube

An Analysis of Deep Neural Networks (arvix)

Capsule Networks – A survey - Journal of King Saud University

Deep learning Yann LeCun 2015