Deep learning assignment -2

Part A:

Subpart -1 (20 marks)

- Implement a baseline CNN, which contains just a single convolutional layer, single pooling layer, fully connected layer and softmax layer.
- Increase the number of layers in your CNN (the number of convolutional and pooling layers). You should implement at least three different CNN configurations (not including the baseline). In your report show the impact on the validation and training accuracy/loss values (inclusive of the baseline case). Compare and contrast the performance of your models in your report.
- Investigate the implementation of data augmentation techniques for two of the above models (please select the two most profound models). In your report, describe the impact(if any), of applying data augmentation on these models. 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?

We have implemented a baseline CNN model that only has

- One convolution layer
- One max-pooling layer
- One fully connected layer
- And finally a softmax over ten classes

We can see the summary of our model in the below image.

There are 16 convolutional filters (3*3) applied with appropriate padding (“same”) such that my input size does not change from the original input image, this helps us deal with corner pixel which can occur less number of times in the output causing it to be neglected. The appropriate padding also ensures that we do not change the image shape. Both vertical and horizontal strides are kept to be 1.

```
Model: "sequential"
_________________________________________________________________
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, 84)                5505108
_________________________________________________________________
dense_1 (Dense)              (None, 17)                1445
=================================================================
Total params: 5,507,001
Trainable params: 5,507,001
Non-trainable params: 0
```

The output layer dimensions can be calculated using the formula given below

d=[(N-K+2P)/S]+1

Where d is the output dimension

N is the original square matrix of image

K is the kernel size

S is the stride assuming equal in both directions (vertical and horizontal)

d=128-3+2+1=128 (meaning the same size as of input)

We also have the depth parameter which will be equal to the total number of filters applied

Second, we use a max-pooling layer with stride two over kernel window of (2*2) effectively halting the image size by now, max-pooling is significantly better than min or average pooling because it helps retain the highest intensity value of the pixel (which in computer vision applications is sometimes used to detect useful edges that are important in describing the essential pieces of the shapes or objects in the image), more on this in the last section of the report

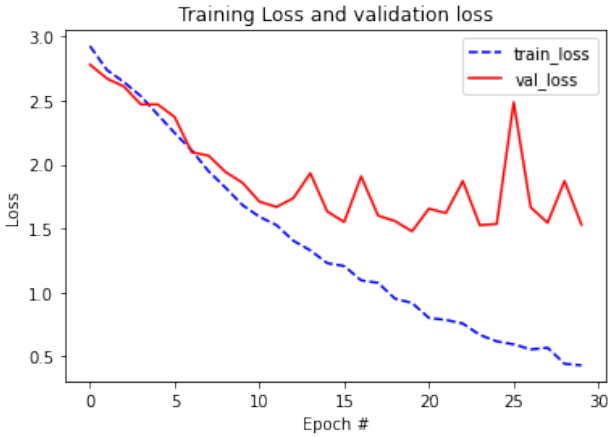
After we have the output, we will then flatten the output and feed this output to a fully connected layer with 84 relu neurons (why 84 ?, because It takes significantly less time with a lower number of neurons to train the network)

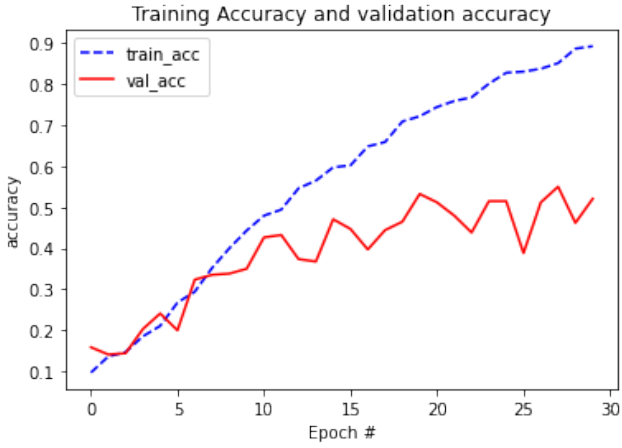
We can choose any number of neurons as we want, but this 84 will yield fast computation, lesser weights and I think it is an ideal size which is not too small neither too large and can be good for a baseline.

(Assignment does not specify how many neurons to place in the baseline, so I will use what I prefer here)

The final layer connected is a 17 class softmax classifier having 17 classes each for an individual category of the flower.

We will be using SGD optimiser with a learning rate of 0.01 and pushing with a batch size of 32 every epoch while monitoring the categorical cross-entropy loss and accuracy over every run of our baseline CNN algorithm.





Please see the graphs attached above as always(previous assignments) the blue dashed line indicates the training and red indicates test performances.

A few subtle observations about the performance achieved are :

- The plot is oscillating slightly this is due to the use of batch size, a smaller batch size means stochastically updating the weights and not for the entire data. If we had pushed the entire data, our line would be very smooth, but this also saves a lot of computation time.

- There is evident overfitting in the data because my model is not performing well on the test data.

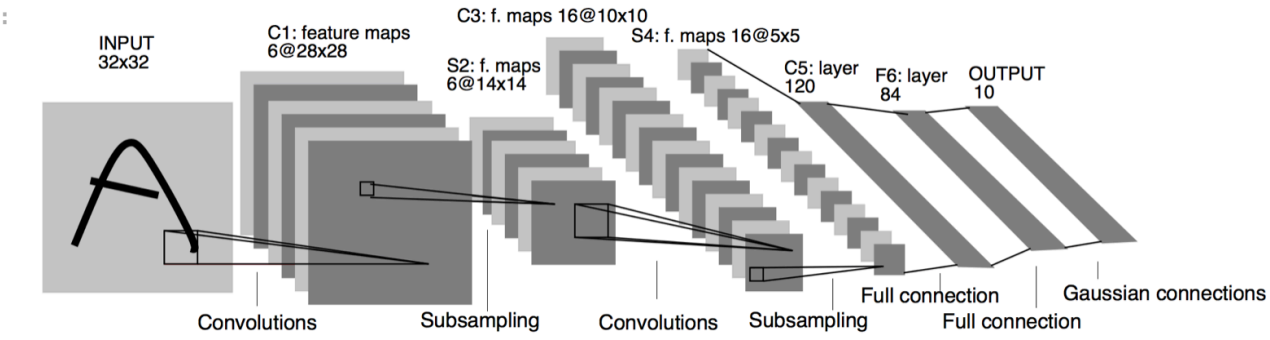
- **The overall accuracy of the baseline model is very low converging at 1.52 loss and 52.06% accuracy on validation data.**

The plots indicate a considerable variation in the performance of train and validation data.

Let us try to build deeper networks and see the impact of overfitting over these deep networks, for the very same reason we will be trying three different neural network architectures.

(Note we are building these architectures from scratch by putting it layer by layer and not importing a direct function of the same, we will also add our modifications to these networks and will list the changes made deviating from what the original author proposed)

The very first architecture we will explore now is the Lenet architecture which is significantly deeper than our baseline CNN and was initially developed by Yan le Can a phenomenal figure in the deep learning domain.



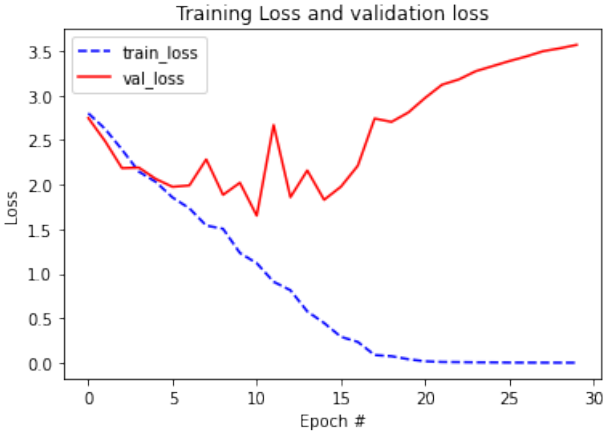
In 1998 he gave this architecture for recognising greyscale images, we will modify this architecture slightly as we will now add relu neurons, to make the comparison with the baseline fairer.

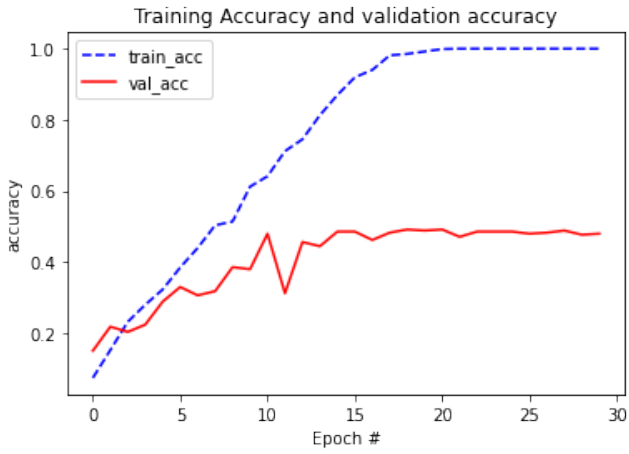
There are just more conv filters with different filter sizes(3*3 and 5*5) and even more fully connected layers in the end, of course, the softmax class will now output 17 as for our flowers problem. The optimiser the learning rate and the batch size is kept the same as before.

From the layer summary given below, we can see that we now have a deeper network with nearly 10 million learning parameter, which is a lot more than the previous 5 million for baseline.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_36 (Conv2D)           (None, 128, 128, 6)       456
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 64, 64, 6)         0
_________________________________________________________________
conv2d_37 (Conv2D)           (None, 64, 64, 16)        2416
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 32, 32, 120)       48120
_________________________________________________________________
flatten_6 (Flatten)          (None, 122880)            0
_________________________________________________________________
dense_17 (Dense)             (None, 84)                10322004
_________________________________________________________________
dense_18 (Dense)             (None, 17)                1445
=================================================================
Total params: 10,374,441
Trainable params: 10,374,441
Non-trainable params: 0
```

Let us now look at the model's performance.





Observations:

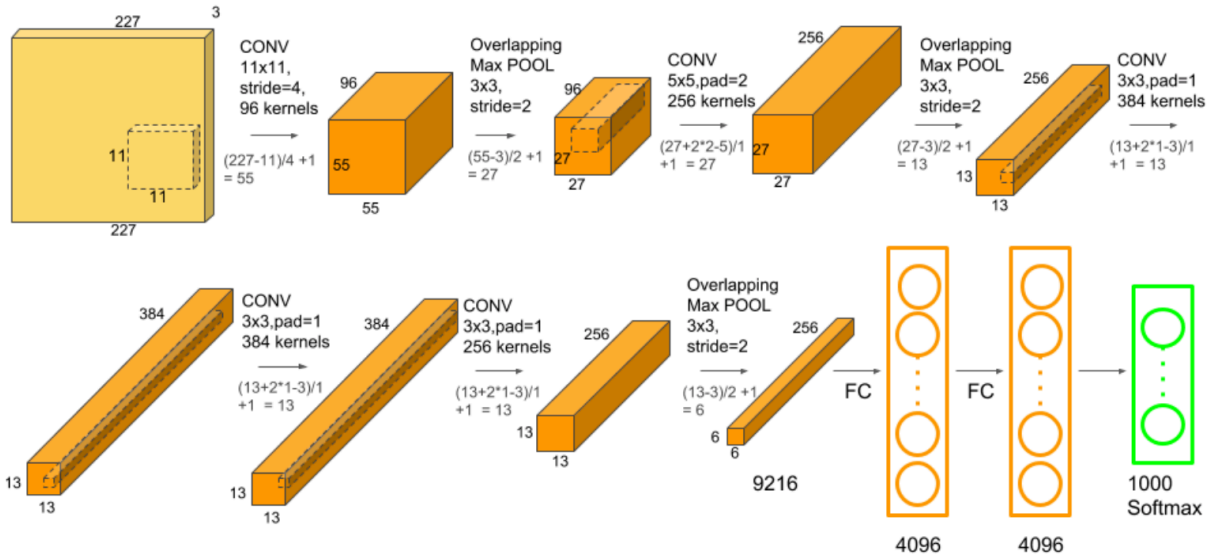
- The model is still overfitting, but the amount of overfitting has significantly increased a lot from the baseline.
- **We are converging (although we can still train as the curve did not flatten but to save time and computational constraints I will avoid running more epochs) at loss 3.56 and 47.94% accuracy which is lower than the baseline.**

(The deeper model has more parameter and thus takes more time to train, despite the fact it is accuracy is much lower than the simple baseline shallow CNN)

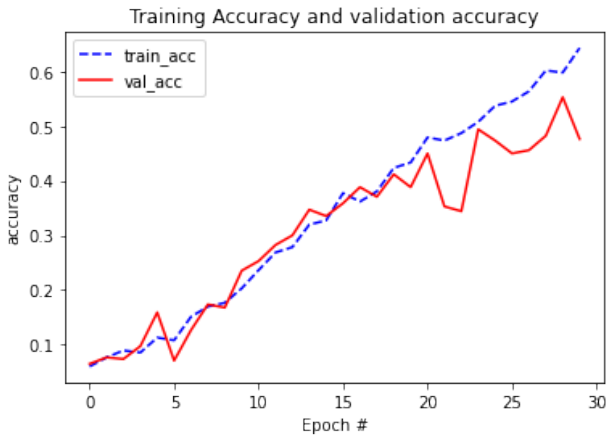
Let us investigate a few more famous and even deeper architectures before we conclude anything.

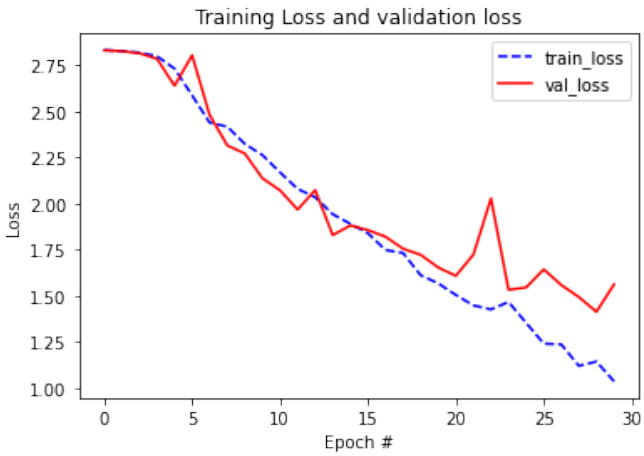
Alexnet architecture:

In the original paper, authors used the concept of Dropout and local response normalisation with Rmsprop optimiser but to keep things constant and to compare we will try to keep the raw architecture design without the use of any of the above mentioned.

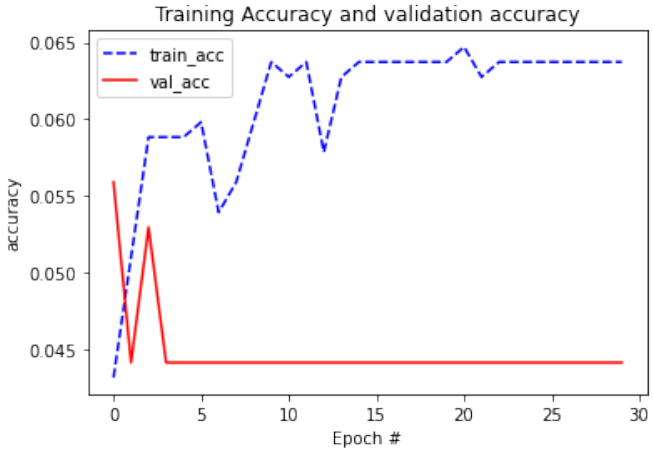


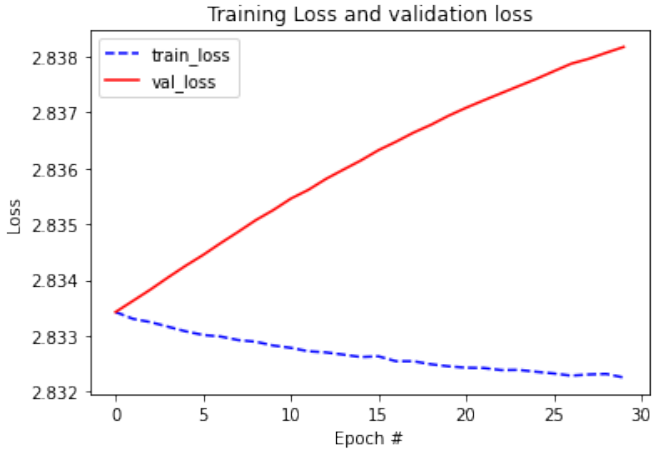
We see the network converge(after 30 epochs) at loss 1.56 and accuracy of 47.65%, However one interesting thing to note is the amount of overfitting has significantly decreased.





VGG-19 network: researchers at Oxford university from a group called Visual Geometry Group proposed this network first. The simplicity of the network outshines Alex net as we had many different types of convolution filters, stride sizes, architecture designs, but VGG is very simple and outperformed Alex net in the Imagenet challenge. Their network proved to outperform Alex net with a very simplistic design (kind of easy to remember two Conv layers followed by a pooling, then three Conv layers followed by pooling)





A severe case of overfitting can be observed with the architecture as the architecture.

It could be due to many reasons:

- The network is far too deep that some layers are not even learning anything.
- The network has no dropouts or batch normalisation.
- Data augmentation not used.
- Activation unit is relu instead of leaky relu
- Batch size could be too small.

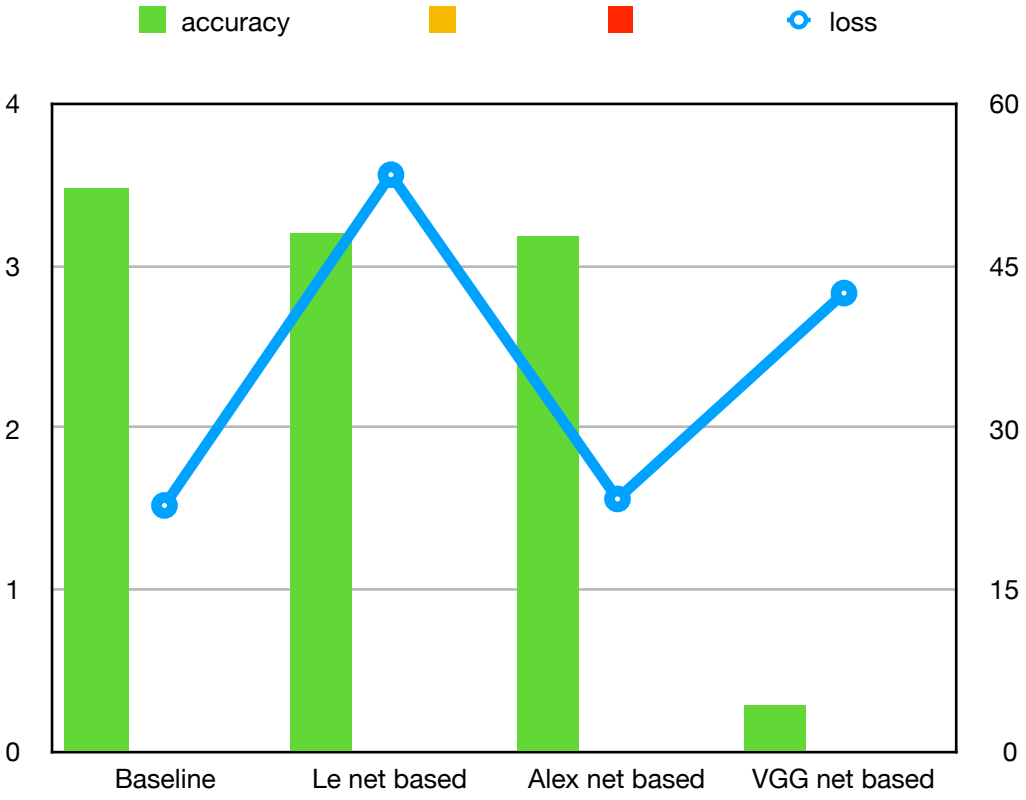
However, to keep things simple, I did not want to incorporate different optimiser, or batch size just wanted to see the impact of the depth.

The accuracy and loss at after 30 epoch was found to be 4% and 2.83 on validation data, which is the worst of all

A summary of these models so far.

Architecture / performance on validation	Loss	Accuracy
Baseline	1.52	52.06
Le net-based	3.56	47.94
Alex net-based	1.56	47.65
VGG net-based	2.83	4.41

The most in-depth networks are performing worst than my baseline, but in a series, we will learn a few techniques that can help me perform better such as the



Data augmentation techniques

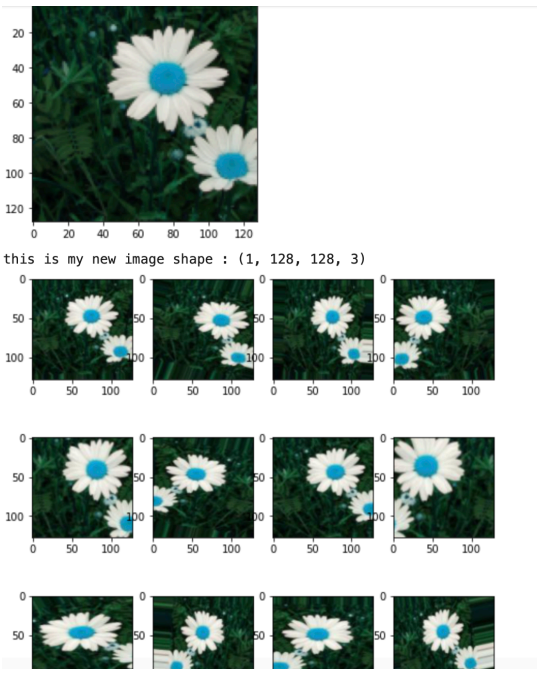
We have built deep CNN's and some they worked well on Imagenet data, but why are not performing well on my flowers data?

The imagenet dataset had many samples of images which means they trained on a much bigger dataset (we just have 80 images of each class of flower). At the same time, the real impact of these architectures can only be concluded from training over them over a large number of samples also our dataset has many variance interns of symmetry and posture of the flowers.

We will try to manipulate the data using data augmentation techniques such as rotations, translations, to be randomly applied over the images before feeding it to the classifier meaning we will now send our batch instance of images that we usually trained for to an augmenter that slightly manipulates these images (adding noise to the image), this technique will make my classifier become more robust shape size or geometry of the object placed. This does not ensure we retain the spatial information (Please see part C of the report for more information) but will still make sure if the object is slightly augmented the classifier can better recognise it. This technique can help us combat the overfitting problem.

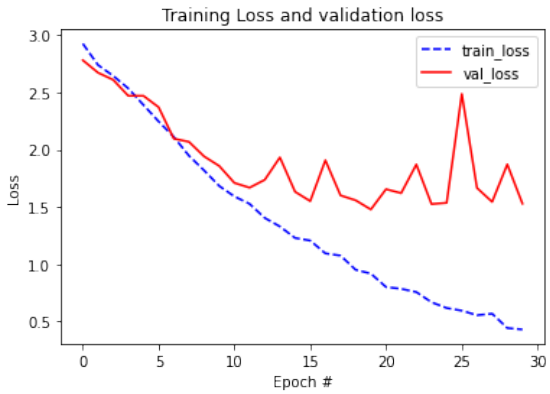
These are some of the images obtained from data augmentation.

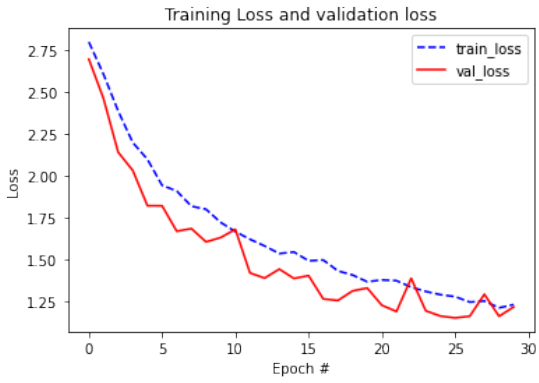
More specifically, we can see how new images are very different from the original in terms of shape size geometry. A random transformation is applied over these images(more information can be found in the code).



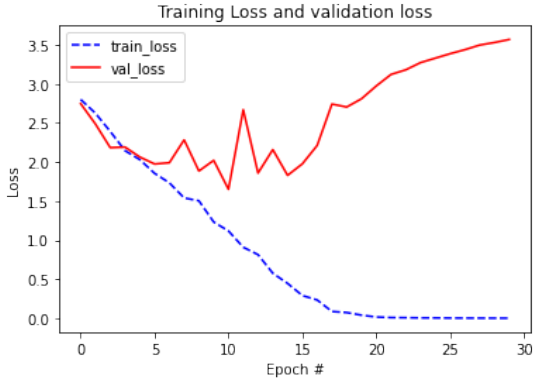
We can see that the overfitting has significantly dropped over all the three networks resulting in better performance than before

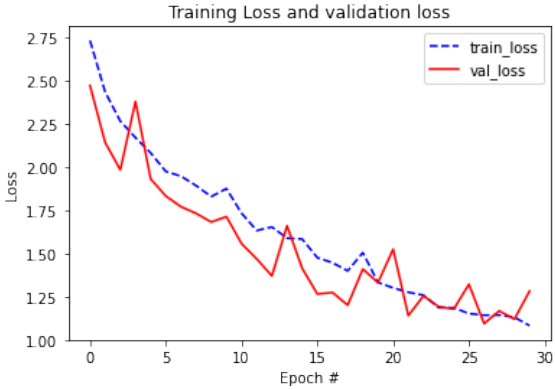
Baseline model (before/ after):



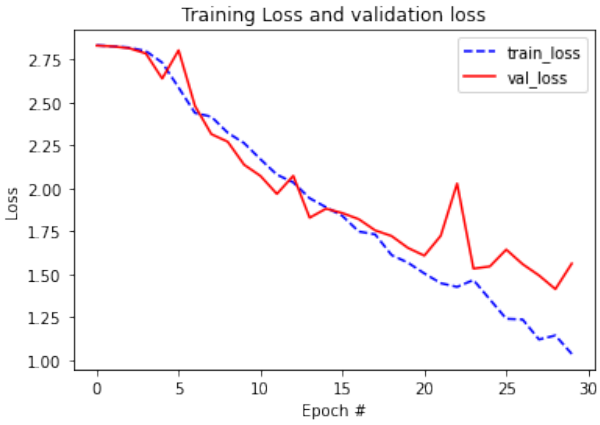


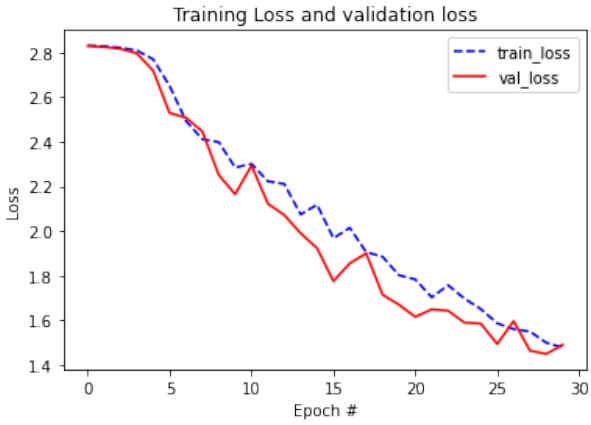
Le net (before/ after):



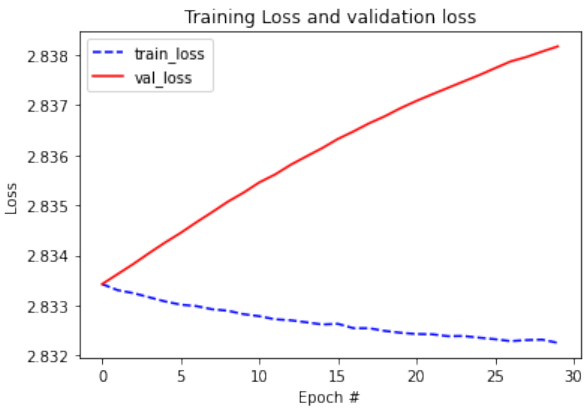


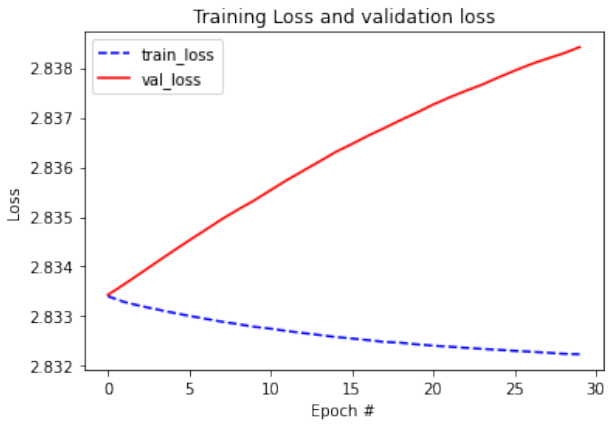
Alex net (before/ after):





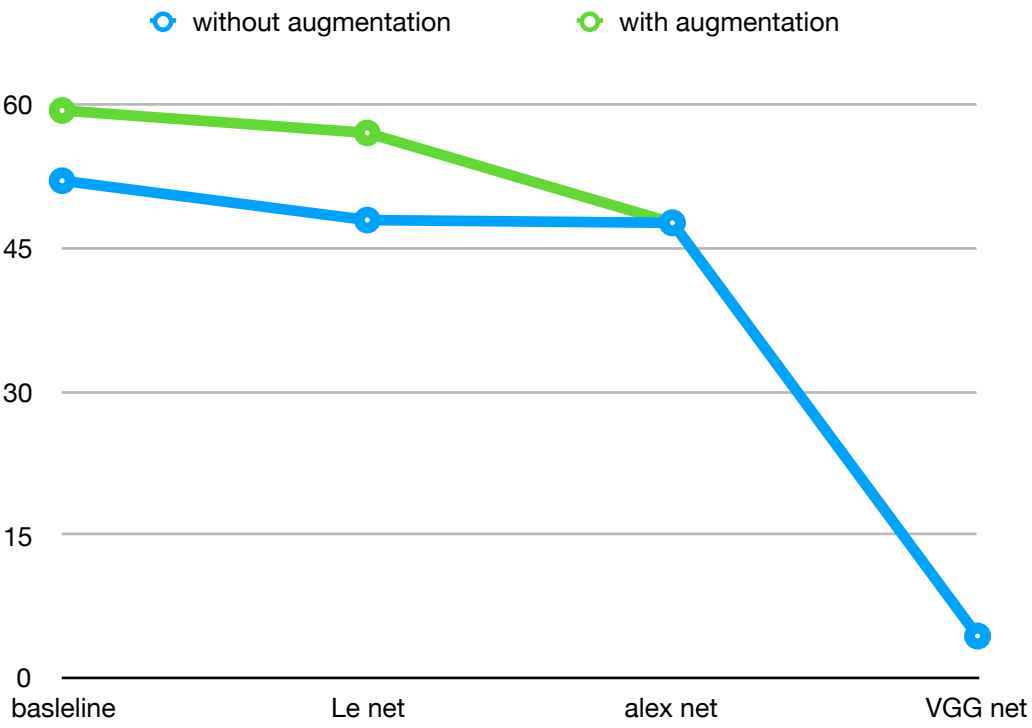
Vgg net (before/after):





To give you a better comprehensive analysis, I will plot a relative loss after augmentation.

Architecture	Loss before augmentation	Loss after augmentation
Baseline	1.52	1.21
Le net-based	3.56	1.28
Alex net-based	1.56	1.48
VGG net-based	2.83	2.83

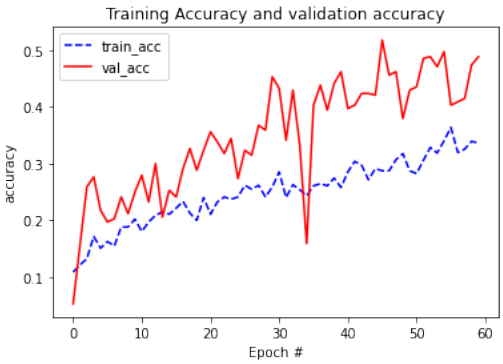


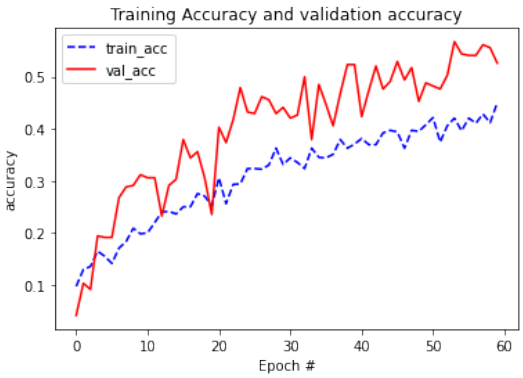
Almost certainly data augmentation helped but for the deeper networks there is something else that is stopping these model from learning, we will specifically now try to tune Alex net and VGG net to see if we can perform better.

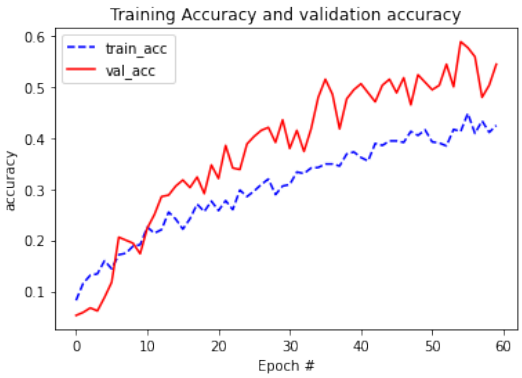
Changes That can improve the deepest two architectures (Alex and VGG)

- We will switch the optimiser to Adam (takes advantage of rms prop and momentum) meaning faster convergence as we saw last time we did not fully converge.
- More number of epochs, this time we will try to double the epochs number to 60 from previous 30 as we are also employing dropouts
- We will now add a concept of dropout that we learnt in the first assignment can significantly improve the robustness of the models acting as another type of regulariser.
- We also use batch normalisation to avoid covariance shift.
- The bigger batch size can help with augmented data but can also lead to substantial changes in the gradients a too small batch size would lead to pointing in the wrong direction of the gradient(will experiment with 16, 32 and 64 batch size)

Batch size 16 vs 32 vs 64

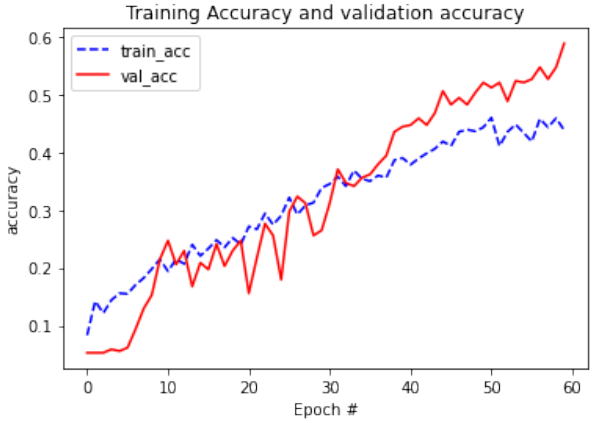






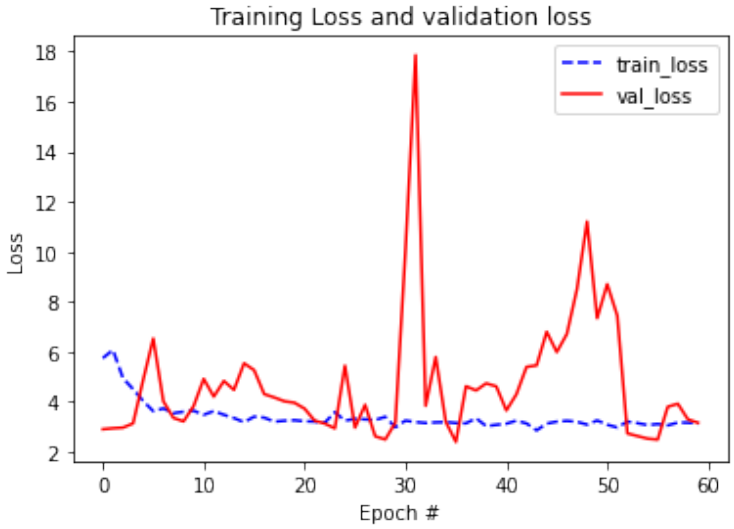
Observation- the batch size of 16 causes heavy oscillations making not an appropriate choice, and 64 has the least amount of oscillations and did not seem to converge

This gives us a hint that maybe 128 batch size would be just right for the network,
The larger batch size can lead to many epochs wasted in the beginning but better in terms of stabilising in the end.



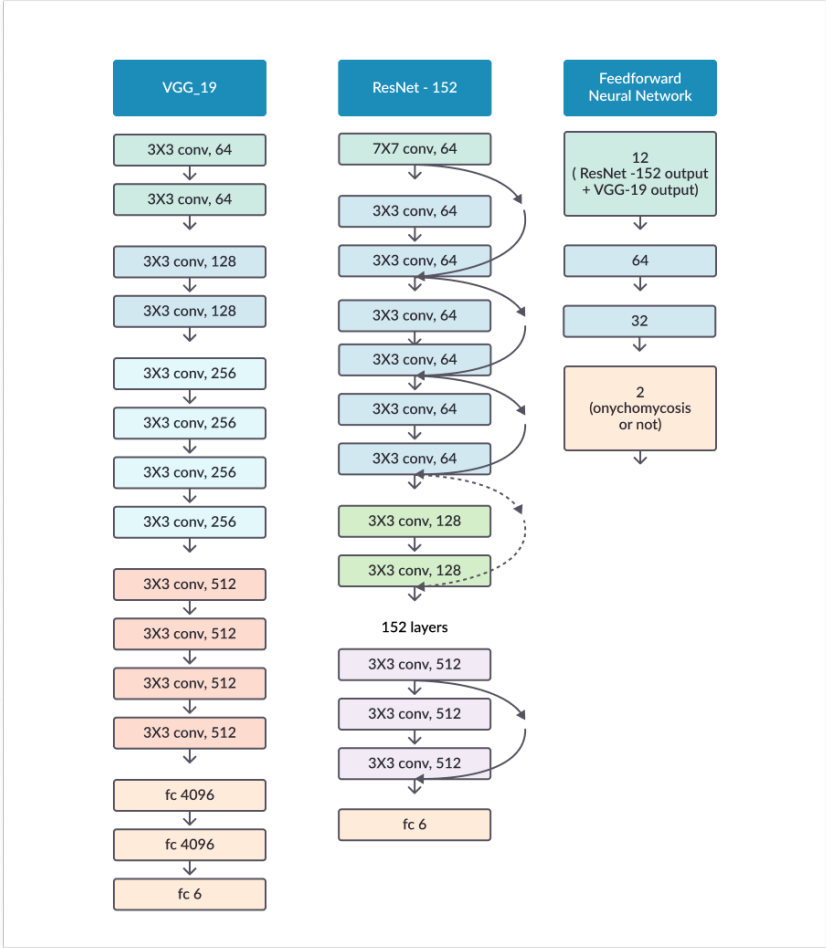
Despite our best efforts trying to tune Alex net, we were only able to push it to a loss of 1.22 and an accuracy of 58.88%

Tuning the parameters helped us increase the accuracy by roughly 5-10 % but not a lot, let us see for the VGG net if they can prove to offer better accuracy.

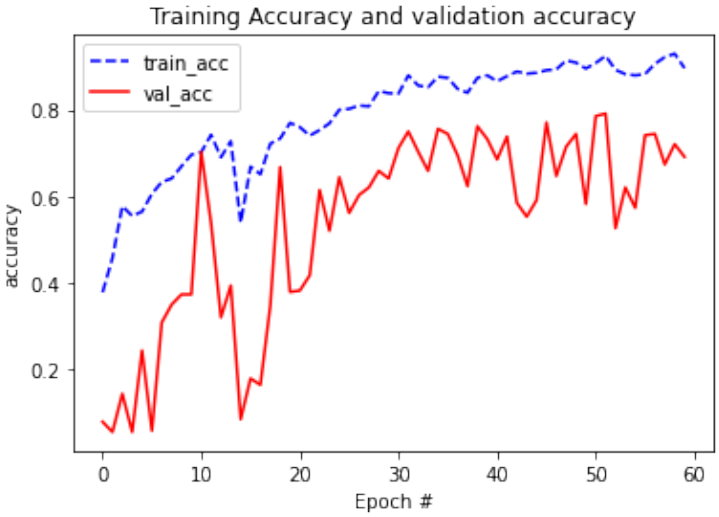


Overfitting was significantly reduced but since layers in VGG are very deep (19 layered some of the layers are dead and not learning. A new proposed architecture by Microsoft in 2015 called Resnet aimed to tackle a similar issue where it used identity mapping after several layers to prevent the layers from getting dead, and in theory, this should perform equally if not better to the shallow counterparts)

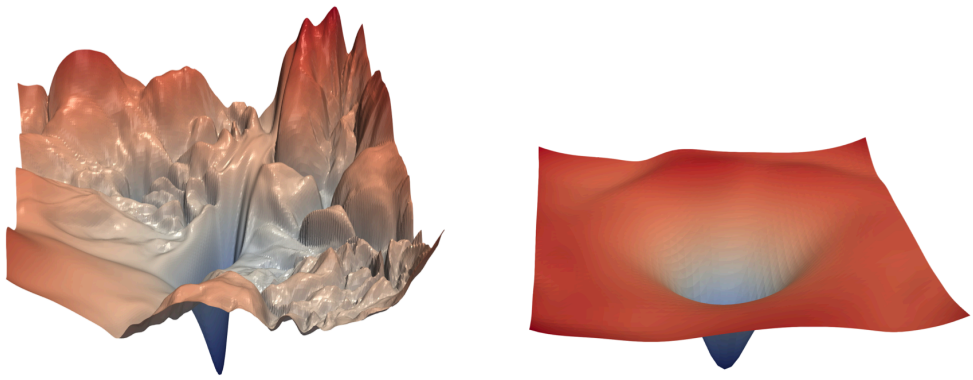
We can try resnet architecture to see the real power of deep neural networks, but for this implementation, I will not build layers from scratch since there are too many layers in the network instead we will import the resent function change the input and output softmax value to suit our flower dataset and then we will set the model to learn. We will experiment with Resnet 50 and resent 152 which can help us perform better on the classification task of flowers since these networks are deep and we have very little data we will set it to a large number of epochs such as 60. Initially, our loss can be too high on the validation, but over time once our network gets tuned, we will have much better results than VGG (plain network). Below diagram indicates the difference between plain and residual networks.



We can see the reset style architecture was able to push through the plain networks resolving in much better accuracies. These results were generated over 64 batch size, which is the reason why the validation plot oscillates a lot. However, using skip connection also ensures that we have a smooth loss function making it easier to propagate and find the global minima. Please see the convex function plot for comparison over skip connection and plain network. We got an accuracy of 80% on rennet 52



Another interesting fact to note is that even with resnets, we are overfitting as there still exists a difference between the accuracies of validation and train data.



(a) without skip connections (b) with skip connections

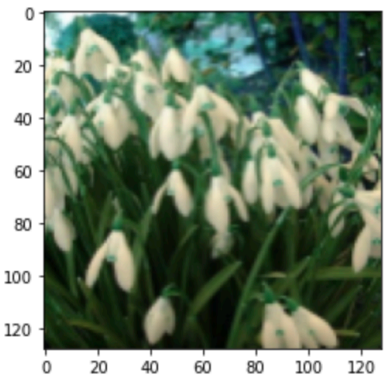
(source of the plot - https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets.pdf)

Also, we could see at many instances(at different epochs in the train test plot for resnet) of our model performed better than final convergence, if it was not trained further. This could be due to batch size pushed (If it had a slightly off image or variation) or could have been the Adam optimiser overshooting the global minima, and trying to get back to the minima point. This could effectively cause oscillations in the loss at multiple standpoints.

Let us save our model and see the performance of resnet-50

```
image=np.expand_dims(image, axis=0)
start=time.time()
o_p=loaded_model.predict(image)
end=time.time()
print("prediction time(in seconds): ",end-start)
print(o_p[0].shape)
print(o_p[0])
label=np.argmax(o_p[0])
chance=(max(o_p[0]))
print("="*20)
print("the predicted label of the image is :{} with confidence :{}".format(label,chance))
print("the actual label of the image is :",testY[num])
```

```
(128, 128, 3)
prediction time(in seconds):  0.09146952629089355
(17,)
[6.4640244e-06 8.5751106e-10 4.9409287e-08 9.5837968e-06 1.9360143e-06
 5.0578547e-06 5.9429453e-06 3.1583550e-08 1.8599143e-04 3.3561710e-07
 3.3381393e-03 2.1910807e-07 9.9643624e-01 1.4973692e-08 5.4268753e-06
 4.4248277e-06 1.2897355e-07]
====================
the predicted label of the image is :12 with confidence :0.9964362382888794
the actual label of the image is : 12
```



For simplicity, I have not saved the previous models because it would take much time to re-run and save the model, but for demonstration, purpose saving one model is okay.

Subpart -2: Ensembles 20 marks

- Implement checkpointing for better model variants
- Implement multiple runs of the same model and save each of the models in a directory

- Design Basic Ensemble model
- Design more sophisticated ensembles

We will now implement checkpointing to save our model to it is the best working state at any of the previous epoch if it had a better performance (meaning lower loss or higher accuracy).

Let us try to experiment this with our baseline CNN model that we made.

We can see that the best version of the model is only saved.

```
Epoch 3/30
31/31 [==============================>.] - ETA: 0s - loss: 2.2734 - accuracy: 0.2368
Epoch 00003: val_accuracy did not improve from 0.26176
32/31 [==============================] - 9s 273ms/step - loss: 2.2668 - accuracy: 0.2373 - val_loss: 2.0508 - val_accurac
0.2618
Epoch 4/30
31/31 [==============================>.] - ETA: 0s - loss: 2.1523 - accuracy: 0.2621
Epoch 00004: val_accuracy improved from 0.26176 to 0.38529, saving model to best_weights_im.hdf5
32/31 [==============================] - 9s 280ms/step - loss: 2.1484 - accuracy: 0.2647 - val_loss: 1.9304 - val_accurac
0.3853
Epoch 5/30
31/31 [==============================>.] - ETA: 0s - loss: 2.0504 - accuracy: 0.3107
Epoch 00005: val_accuracy improved from 0.38529 to 0.43529, saving model to best_weights_im.hdf5
32/31 [==============================] - 9s 280ms/step - loss: 2.0443 - accuracy: 0.3127 - val_loss: 1.8310 - val_accurac
0.4353
Epoch 6/30
31/31 [==============================>.] - ETA: 0s - loss: 1.9420 - accuracy: 0.3543
Epoch 00006: val_accuracy did not improve from 0.43529
32/31 [==============================] - 9s 271ms/step - loss: 1.9416 - accuracy: 0.3500 - val_loss: 1.7851 - val_accurac
0.4029
Epoch 7/30
31/31 [==============================>.] - ETA: 0s - loss: 1.8786 - accuracy: 0.3502
Epoch 00007: val_accuracy improved from 0.43529 to 0.44118, saving model to best_weights_im.hdf5
32/31 [==============================] - 9s 282ms/step - loss: 1.8764 - accuracy: 0.3529 - val_loss: 1.7077 - val_accurac
0.4412
Epoch 8/30
31/31 [==============================>.] - ETA: 0s - loss: 1.8406 - accuracy: 0.3664
Epoch 00008: val_accuracy improved from 0.44118 to 0.45588, saving model to best_weights_im.hdf5
32/31 [==============================] - 9s 278ms/step - loss: 1.8332 - accuracy: 0.3745 - val_loss: 1.5802 - val_accurac
0.4559
Epoch 9/30
31/31 [==============================>.] - ETA: 0s - loss: 1.7746 - accuracy: 0.3988
```

Let us now look at making ensemble models.

To first make an ensemble model, we will save multiple base learners.

The quality of ensemble depends upon the variability of these models. The more different these base learners are, the better the ensemble predictions.

For a simple case let us freeze a particular architecture, say baseline or le net. Then we try to reinitialise the model since the model by default draws weights from a random distribution each time after a fixed number of epoch my weights will be slightly different for each of the models, this introduces variability in base learners. However, it is not a very good variability to make an ensemble in the future sections of the report we will experiment on better ensembles.

To execute this ensemble model, we will also be using the checkpointing function that returns us the best epoch of individual base learners. This makes sure that each of the base learners we save is at it is best checkpoint or epoch.

To ease this task, we made a function called make_ensemble

```
make_ensemble(model,trainX,trainY,testX,testY,name_of_this_model="baseline",epochs=30,
                      return_model=True,relate_by="loss",base_learner_count=5)
```

We pass our model for which we want the ensembles to be made.

The name of the model for which there will be directory made
The number of base learners by default is five, but you can change to any value (I put this to save computation time)

If return_model is set to True, this function will return you the model(stored in a JSON file) the weights (in h5 file) and the model title which we defined.

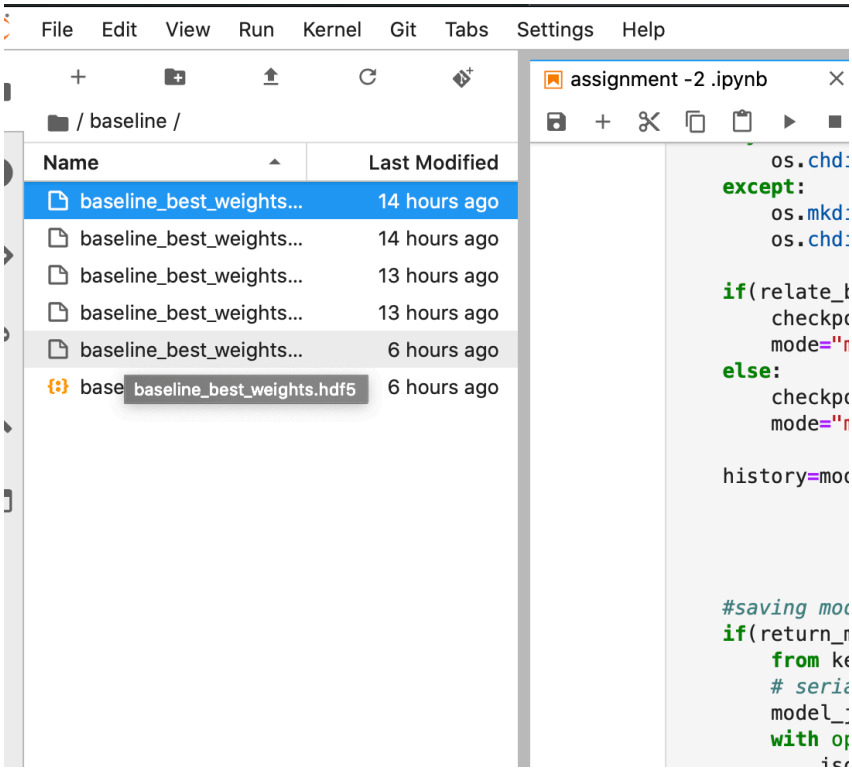
Relate by is a loss, meaning it will checkpoint the lowest loss model else it will checkpoint by highest accuracy.

This function internally calls a checkpointing model.

```
def check_pointing_model(model,trainX,trainY,testX,testY,name_of_this_model="default_model",epochs=30,
                         return_model=True,relate_by="loss",fname = "best_weights.hdf5"):
    """this function take a cnn model and returns the best checkpoint of model weights"""
```

Once we run this,

We can see in our specified model directory all the base learners saved to their best checkpoint.



To evaluate the baselines, we will call evaluate ensemble function, providing it with the directory or a list of directories where our ensemble models are stored and the data to evaluate our models for.

```
base_line_dir="/home/jupyter/baseline"
evaluate_ensembles(base_line_dir,testX,testY)
```

If testY is given, it will evaluate the ensemble models over the testing data, and if it is absent, it will just return the ensemble model's prediction.

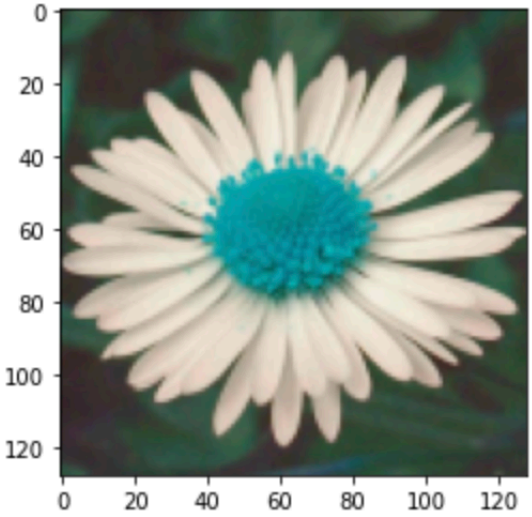
Since we had five models and a single directory, we can now see the output from each of the model and its performance on the testing data.

```
accuracy: 64.71%
loss : 1.0254183657029097
[1.0254183657029097, 0.6470588445663452]
accuracy: 63.82%
loss : 1.1187339824788711
[1.1187339824788711, 0.6382352709770203]
accuracy: 50.00%
loss : 1.454742748597089
[1.454742748597089, 0.5]
accuracy: 69.71%
loss : 0.9490555847392362
[0.9490555847392362, 0.6970587968826294]
accuracy: 68.53%
loss : 0.9684233174604528
[0.9684233174604528, 0.6852940917015076]
(340,)
ensemble accuracy: 69.41176470588235 %
```

We can also see that ensemble model of 5 learners provides slightly higher than the average accuracy of each of these models but not more than the maximum accuracy of 4th model this is because there is not a lot variability in base learners.

We can also pass a single input with any labels to see what our ensemble model thinks of this input, below is a screenshot of the same indicating probability with a confidence score.

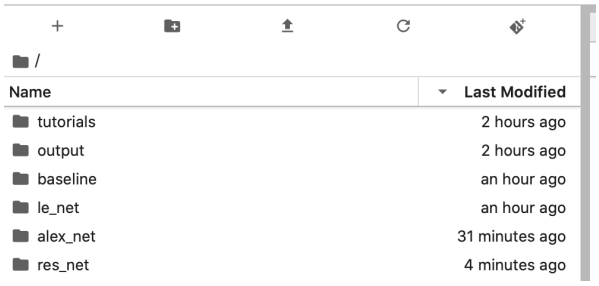
```
(1,)
predictions: [6]
prediction time(in seconds):  1.0054121017456055
====================
the predicted label of the image is :6 with confidence :0.9989784955978394
the actual label of the image is : 6
```



Just like the baseline, we can train to create multiple base learners for each of the networks we trained before, while also adding the advantage of the checkpointing and the data augmentation.

To enable variability we will create a directory of each architecture type that we created except VGG because it gave the last performance, so we now created four directories.

- 5- base learners of baseline CNN's
- 5- base learners of Le-net CNN's
- 5-base learners of Alex net CNN's
- 5-base learners of Resnet-52 CNN's



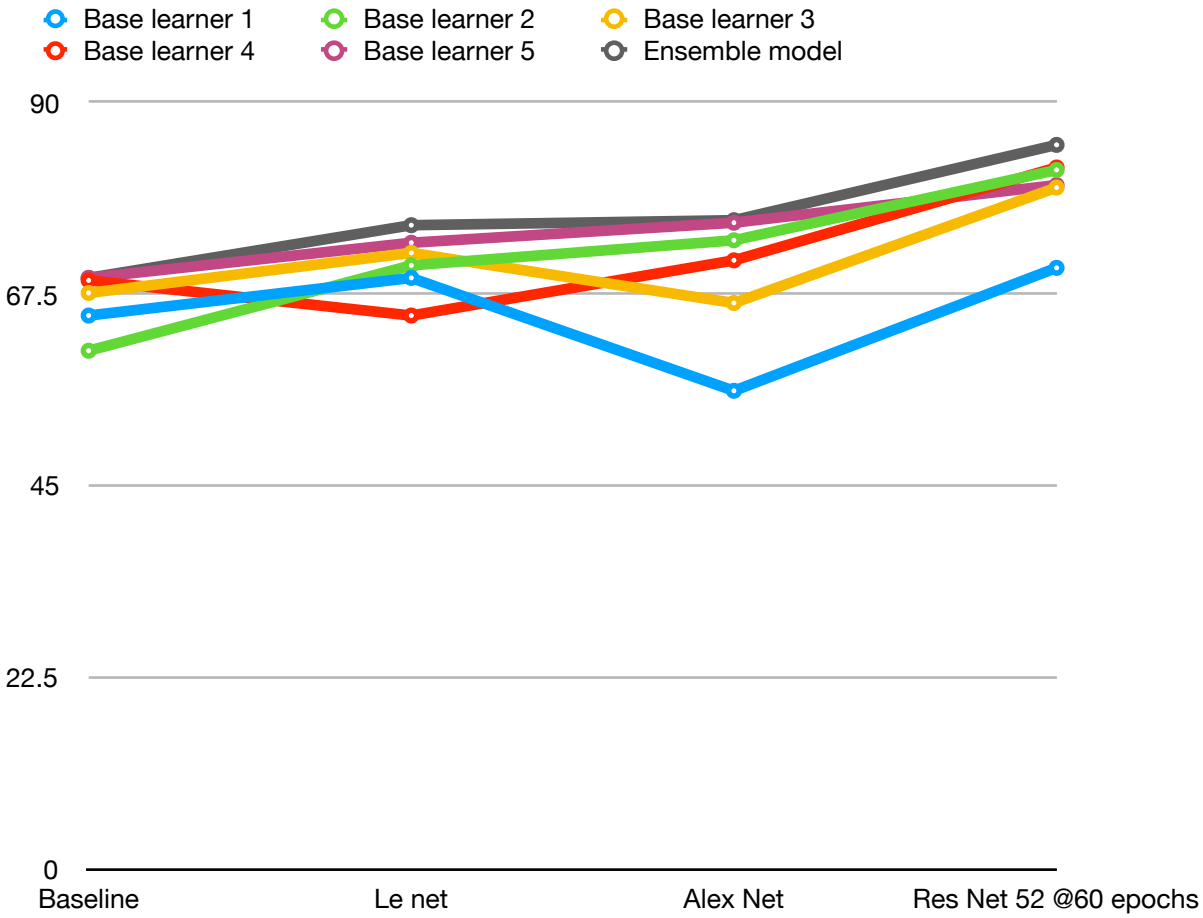
Let us compare the performance of Ensemble model with respect tot he base learners on the testing dataset.

	Baseline	Le net	Alex Net	Res Net 52 @60 epochs
Base learner 1	65	69.41	56.18	70.59
Base learner 2	60.88	70.88	73.82	82.06
Base learner 3	67.65	72.35	66.47	80.00
Base learner 4	69.12	65	71.47	82.35
Base learner 5	69.41	73.53	75.88	80.29
Ensemble model	69.41	75.588	76.17	85.00

An example of ensemble Alex net output from Console

```
accuracy: 56.18%
loss : 1.3106908671996174
[1.3106908671996174, 0.5617647171020508]
accuracy: 73.82%
loss : 0.8939186075154473
[0.8939186075154473, 0.7382352948188782]
accuracy: 66.47%
loss : 0.9729652909671559
[0.9729652909671559, 0.6647058725357056]
accuracy: 71.47%
loss : 0.8923128001830157
[0.8923128001830157, 0.7147058844566345]
accuracy: 75.88%
loss : 1.0552129801581889
[1.0552129801581889, 0.7588235139846802]
(340,)
ensemble accuracy: 76.17647058823529 %
```

The ensemble of extreme variance model leads to a significant boost in the accuracy, meaning each of the base learners learns a different aspect of the data. This I an effortless averaging Ensemble that we have implemented. However, there are many complex types of ensembles available which we will see later on.



To add variance to Ensemble models, we will select a best of each architecture (out of 5 base learners in each of the folder) and then run it together; we observe roughly 1-5% increase performance for the same model. If the variance is too much, we might perform even worse than the best model as in the case of variance_ensemble where we get an accuracy of 80% while one of the resent models perform 82.35 this is because other weak learners are performing very poorly as compared to the best weak learner, Simple Ensembles generally tend to bias the majority confidence classifiers. Let us try to look if we can make a better ensembling technique.

```
#==>> this is baseline

model_1 = tf.keras.wrappers.scikit_learn.KerasClassifier(build_fn=model_1, batch_size=16)


#this is the le net f
model_2 = tf.keras.wrappers.scikit_learn.KerasClassifier(build_fn=model_2, batch_size=16)

#this is the alex net
model_3 = tf.keras.wrappers.scikit_learn.KerasClassifier(build_fn=model_3, batch_size=16)
#this is the res_net
model_4 = tf.keras.wrappers.scikit_learn.KerasClassifier(build_fn=model_4, batch_size=16)



lr = LogisticRegression()



#make the stacking classifier

sclf = StackingClassifier(classifiers=[model_1, model_2, model_3,model_4],
                          meta_classifier=lr)

for clf, label in zip([model_1, model_2, model_3, model_4,sclf],
                      ['base_line',
                       'le_net',
                       'alex_net',
                       "res_net 52",
                       'StackingClassifier']):

    scores = model_selection.cross_val_score(clf, testX, testY,
                                             cv=3, scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

Stacking classifier for ensemble (used Keras - scikit learn wrapper).

By using stacking classifier we are training another meta model on the outputs received from the neural net based models or the base learners this output is then feed to the logistic regression classifier which is further tuned to give even better accuracies.

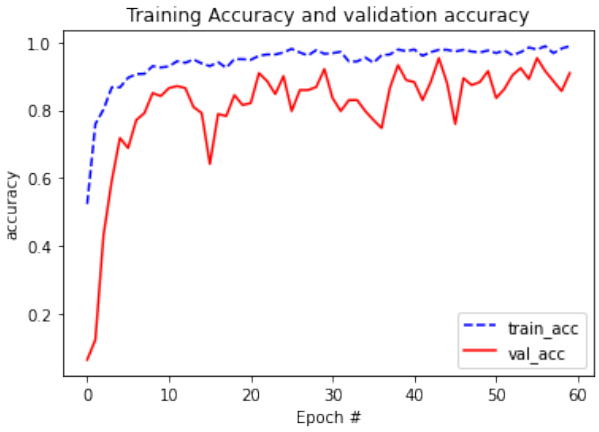
Part B:

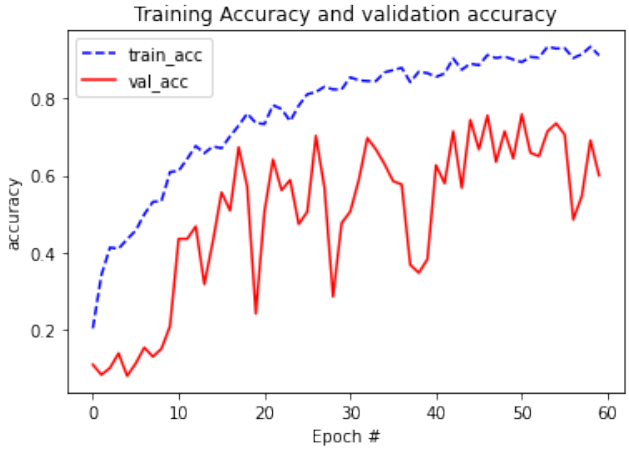
We will now reuse the model that we trained in Part A for resnet where we initialised our weight to be None, let us see the impact when we now initialise our weights to be that of Imagenet.

	Res net 50	Res net 152 V2	Dense Net 121	Dense net 201
With image net weights	93.52%	92.05%	92.64%	95.294%
With None weights	80%	74.70%	80.88%	75.882%

These model are generated with data augmentation and checkpointing as used previously.

We see that generally, Image net weight initialisation works well in faster learning and in terms of performance.





Figures above indicate DenseNet performance over different weights initialisation:

Image net weights yield better accuracy for the same number of epochs on the validation data.

Feature extraction: to use feature extraction with our data, we will use Dense net 201 architecture that previously gave us the best performance. We will use the image transformation as with the image net weights, and then we will plug it in our ML model instead of fully connected layers.

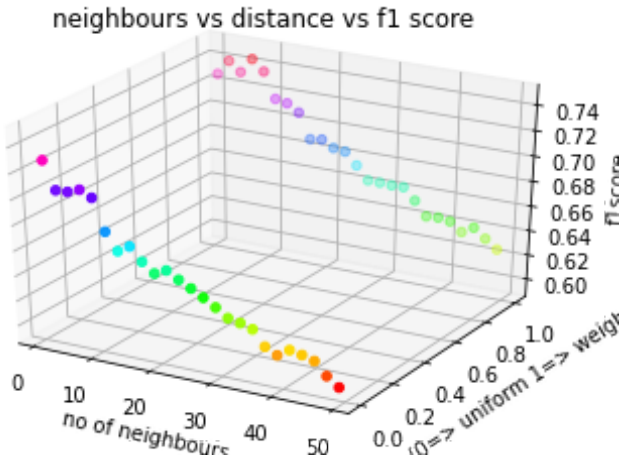
We could have tried for all the different architectures plugging them to an ML model but this was computationally expensive so I tried with only Densenet that was performing reasonably well before.

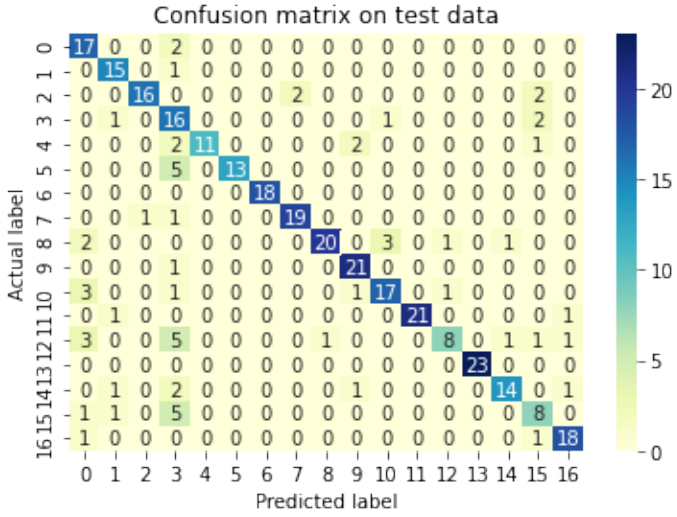
In the next part for fine tuning we try an even better architecture called Inceptionresnet V2 which has 2% increase in top 1% error on image net when compared with Densenet.

	Knn (optimal= K=5, distance)	Logistic regression	Random forest	Support Vector machines with RBF kernels
With tuning	80.84%	90.88%	62.64	0.9
Without tuning	80%	91.17	-	-

There is not much difference when comparing the tuned ML model for feature extractor to the untuned model.

Knn tuning for the right number k neighbours, the weights (whether to take uniform KNN or the weighted KNN), finally the performance of the model in terms of F1 score can be seen in the 3d plot, while the confusion matrix indicate that class 3 has the highest conflicts (maybe we need more and better samples for this class)





Fine tuning:

For fine-tuning, the model we are taking is **inceptionresnetV2 (as it performed well on Imagnet data even better than Dense net)** which is already performing with a 95% accuracy on the validation data, using this technique should help us further fine-tune the image net weights to get even better accuracies than before.

We could have tried keep different layers trainable and non trainable but again considering the computations I will try with only 6 configurations.

Validation with 64 batch size	0% not trainable	10% not trainable	20 % not trainable	30% not trainable	40% not trainable	50% not trainable	60% not trainable
Accuracy	95.29%	95.58%	96.76%	95.88%	96.76	96.47%	95.58
loss	-	0.24	0.1677	0.22	0.24	0.32	0.36

Zero fine-tuning is meaning we are using image net weights entirely.

We see that when keeping a lesser amount of non-trainable layers, we get better accuracy to let us try to increase the batch size to see if we can yield even better accuracy for same non-trainable layers.

Validation	10% not trainable	20% not trainable
Small batch size of 16	95.52%	95.00
A small batch of 32	95.58	94.70
A large batch of 128	95.58	95.00
Large batch size of 256	96.176%	95.88

We notice that somewhere keeping the network weights from imagenet to be known trainable between (20-40%) can yield us some good accuracies and a batch size of (32-256) should give us an accuracy of 96.76% at the maximum while 95% as the minimum.

Part C: Research based on capsule networks

Dynamic Routing between Capsules:

CNN's generally work by accumulating sets of features at each layer. It starts by finding edges, then shapes, then actual objects. However, the relationship between the Spatio-temporal features is lost while using CNNs. For example, while identifying a face, a mere presence of two eyes, one nose and one mouth are enough for a CNN to classify it as a face, the presence of these objects in the wrong place does not affect the accuracy of the CNNs. As per Geoffrey Hinton et. al. “The pooling operation used in convolutional neural networks is a big mistake, and the fact that it works so well is a disaster.” The introduction of the capsule network gives us the capability to take advantage of Spatio-temporal relationship between features.

As per Sara Sabour et al. [], the output vector of the capsule represents the probability of input capsule content. For this, they used a non-linear ‘squashing’ function, which gives the probability of each capsule in a range of 0 to 1 as defined in eq.

$$P_j = \frac{\| C_j \|^2}{1+\| C_j \|^2} \frac{C_j}{\| C_j \|^2}$$

$$C_j = \sum k_{ij} \hat{u}_{i|j}$$

$$\hat{u}_{i|j} = W_{ij} u_i$$

$$k_{ij} = \frac{\exp(b_{ij})}{\sum_a (b_{ia})}$$

Where P_j is the vector output of the capsule j and C_j is its total input?

Where C_j is the weighted sum of all the prediction vector $\hat{u}_{i|j}$ taken from the capsules in the layer below, and these prediction vectors can be calculated by multiplying the output u_i by the weight matrix W_{ij}

k_{ij} ?

Where is the coupling coefficient calculated by running the dynamic routing process iteratively?

The sum of all the coupling coefficient always equals to 1. These coupling coefficients are determined using “routing SoftMax”, log prior probability representing the coupling between the capsule I and capsule j is represented by b_{ij} .

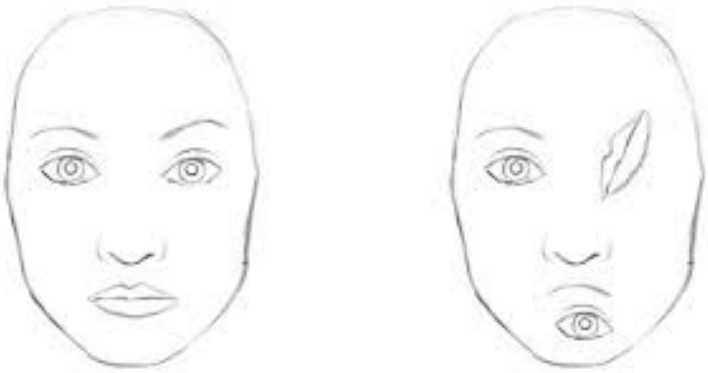
Similar to all the other weights, log priors can also be learned. They are dependent on the location and type of capsules. Nonetheless, they do not depend on the current input image.

The agreement between the current output P_j of each capsule j in the above layer and prediction vector $\hat{u}_{i|j}$ made by capsule I am iteratively measured for defining initial coupling coefficient.

The agreement is nothing but the simple scalar product $a_{ij} = P_j \cdot \hat{u}_{i|j}$. This agreement can be treated as it is same as the log and added with initial logit, b_{ij} before the computing new values of all the coupling coefficient linking capsule I over the higher-level capsule.

In convolutional capsule layers, the output of each capsule is nothing but the local grid of vectors to each type of capsules in the above layer which can be calculated using different matrix transformation for each member of the grid and each type of capsule.

The below diagram illustrates how the CNN vs a capsule network look at a particular image and why CNN’s can be fooled, whereas capsule taking the symmetry of the shape in the account are pretty hard to fool.



A different shifting variance of sub-object such as eyes and nose.

Various Angles, tilt in the object.

Depth of view from different angles indicating different geometry of the same/ similar object

CNN's cannot differentiate between the two pictures because of the pooling operation used as it merely detects the highest edge anywhere present in the image which can be detected meaning if anywhere we have a unique feature with high-intensity order it is detected. However, the shape or the symmetry or the orientation of the pixels are not taken into account by the classical CNN models with pooling.

This helps us resolve the corner cases such as in MNIST data where Classical CNN's have trouble identifying between the six and the 9's.

References-

1. Dynamic Routing Between Capsules by Sara Sabour, Nicholas Frosst, Geoffrey E Hinton
2. https://medium.com/ai³-theory-practice-business/understanding-Hinton's-capsule-networks-part-i-intuition-b4b559d1159b