Deep Learning

Assignment 2

Rajbir Bhattacharjee

R00195734

Contents

[Convolutional Neural Networks 3](#_Toc70897331)

[Implementing a CNN 3](#_Toc70897332)

[Implementation Details and Common Functions 3](#_Toc70897333)

[Baseline CNN 3](#_Toc70897334)

[One Covolution and Two Dense Layers 6](#_Toc70897335)

[Two Convolutional Layers and One Dense Layer 10](#_Toc70897336)

[Two convolutional and two dense layers 13](#_Toc70897337)

[Three convolutional and two dense layers 16](#_Toc70897338)

[Three convolutional and two dense layers with dropout 18](#_Toc70897339)

[Comparing all previous runs 20](#_Toc70897340)

[Data augmentation 23](#_Toc70897341)

[Data augmentation on three convolutional and two dense layers with dropout 23](#_Toc70897342)

[Data augmentation on two convolutional and two dense layers 29](#_Toc70897343)

[Few Shot Learning 33](#_Toc70897344)

[One shot learning 33](#_Toc70897345)

[Zero Shot Learning 35](#_Toc70897346)

[Theoretical underpinning of few-shot learning 35](#_Toc70897347)

[Ensembles 37](#_Toc70897348)

[Methodology 40](#_Toc70897349)

[Need for Variability 41](#_Toc70897350)

[Results 43](#_Toc70897351)

[Future work 46](#_Toc70897352)

[Transfer Learning 46](#_Toc70897353)

[Pre-trained networks as feature extractors 46](#_Toc70897354)

[Modify and Finetune 48](#_Toc70897355)

[Capsule Networks 52](#_Toc70897356)

# Convolutional Neural Networks

## Implementing a CNN

### Implementation Details and Common Functions

Several CNN’s were implemented and early stopping was introduced. In some of the experiments, the learning rate was decayed as the training progressed. Experiments were also performed by decaying the learning rate when a plateau was hit, but since this was taking too long to converge, they were not followed through.

It was found that Colab could run out of memory while training a neural network. It was observed that manually calling Python’s garbage collector mitigated some of that, but for long training durations the problem still occurred. A trick to call Python’s garbage collector within the training cycle helped mitigate this. The learning rate callback function was piggybacked to force garbage collection as this would be called on each epoch. After implementation of this, the system never ran out of memory.

**def** decay\_after\_runs**(**N**):**

NRUNS **=** N

# We're sneaking this in, since this will be called at every epoch

# it also gives us a good way to force calls to gc() within the fit function

**def** learning\_rate\_scheduler**(**epoch**,** lr**):**

gc**.**collect**()**

**if** NRUNS **<** 0 **or** epoch **<** NRUNS**:**

**return** lr

**else:**

**print(**f"Learning Rate: {lr} --> {lr **\*** tf**.**math**.**exp**(-**1.0**)**}"**)**

**return** lr **\*** tf**.**math**.**exp**(-**1.0**)**

**return** learning\_rate\_scheduler

lr\_callback **=** tf**.**keras**.**callbacks**.**LearningRateScheduler**(**decay\_after\_runs**(**20**))**

The images were normalized by dividing by 255, and then passed through the CNN models. Several models of CNN’s were tried.

### Baseline CNN

The baseline CNN was implemented as follows:

**class** **BaselineShallowNet:**

*@staticmethod*

**def** build**(**inshape**,** nlabels**):**

inputShape **=** trainX**[**0**].**shape

model **=** tf**.**keras**.**Sequential**(**name**=**"BaseLineShallowNet"**)**

model**.**add**(**tf**.**keras**.**layers**.**Conv2D**(**64**,** **(**3**,**3**),** padding**=**'same'**,** \

input\_shape**=**inshape**,** activation**=**'relu'**))**

model**.**add**(**tf**.**keras**.**layers**.**MaxPooling2D**(**2**,** 2**,))**

model**.**add**(**tf**.**keras**.**layers**.**Flatten**())**

model**.**add**(**tf**.**keras**.**layers**.**Dense**(**nlabels**,** activation**=**'softmax'**))**

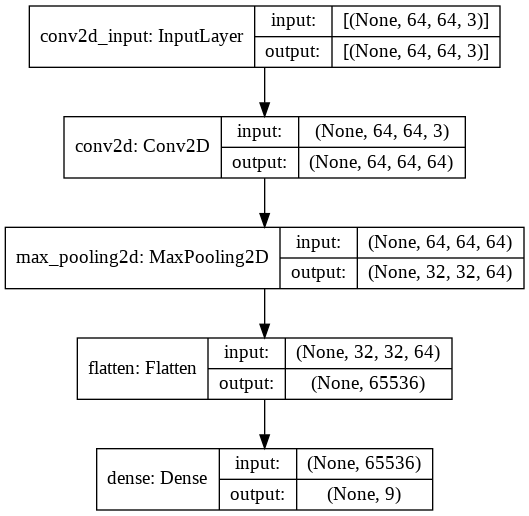
plot **=** tf**.**keras**.**utils**.**plot\_model**(**model**,** show\_shapes**=True,** \

expand\_nested**=True)**

IPython**.**display**.**display**(**plot**)**

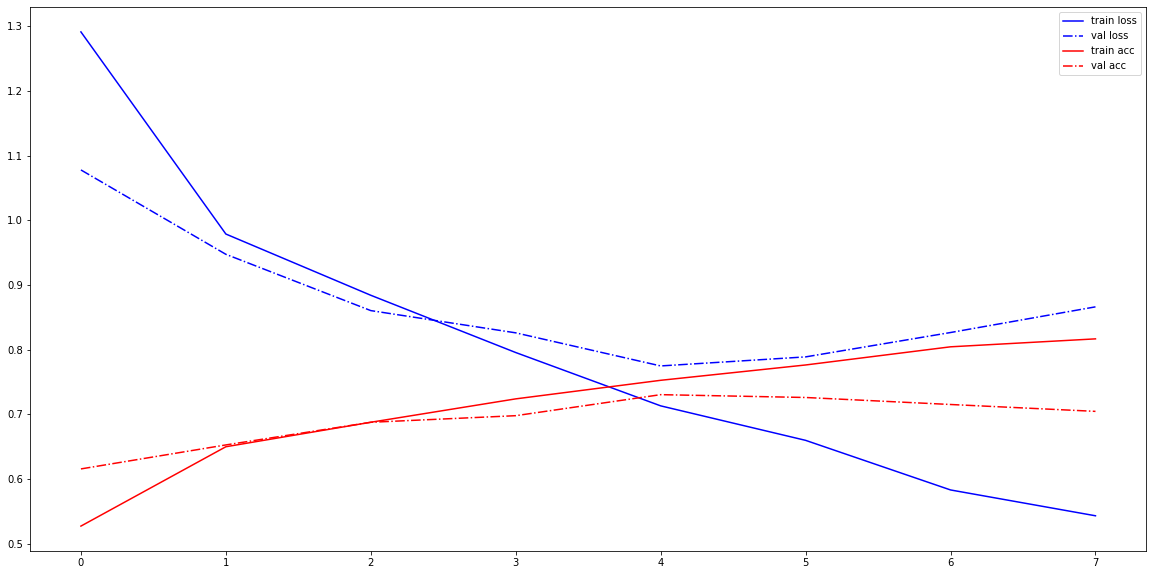
**print(**model**.**summary**())**

**return** model



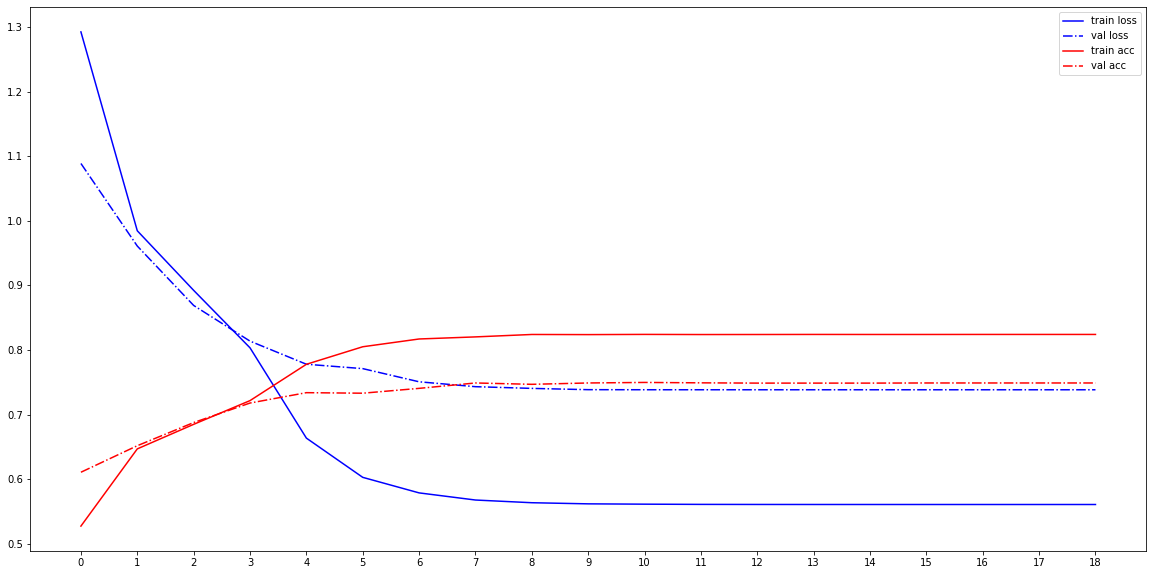
The number of trainable parameters for this model were 591,625, and most of these were in the dense layer.

The results of training this model were as follows:

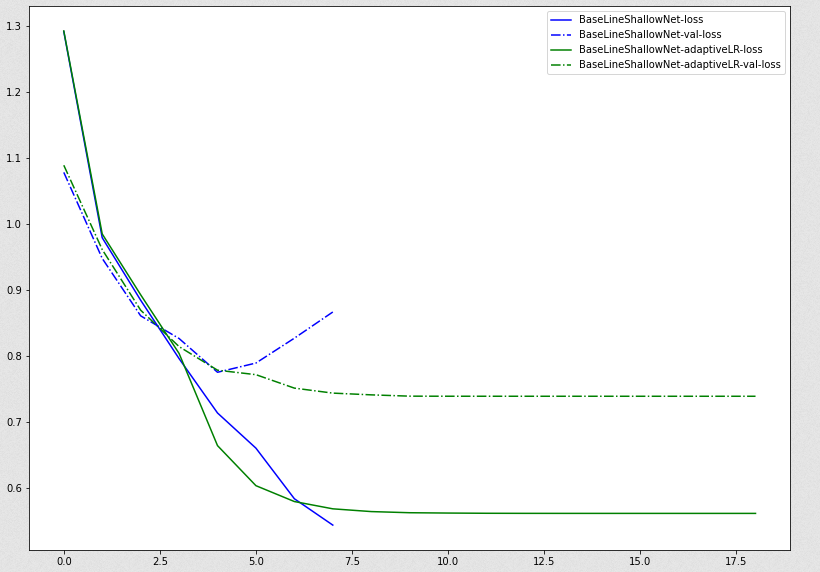


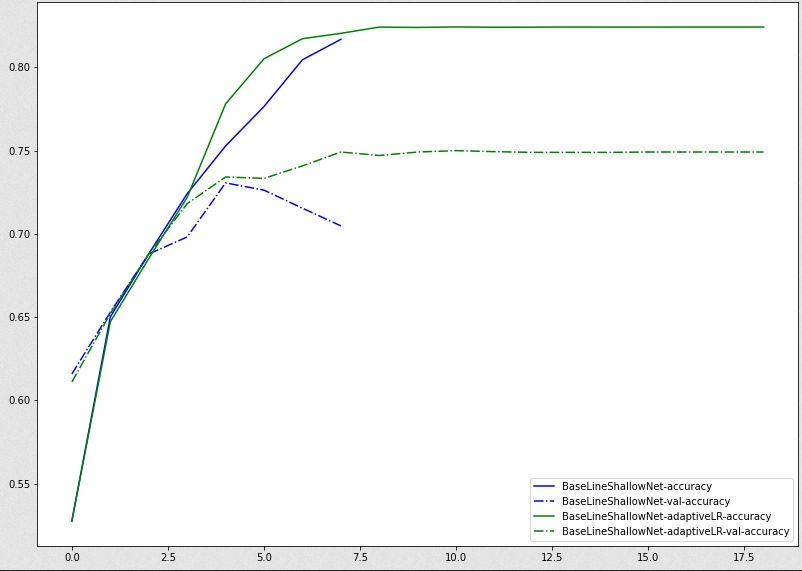
As we can see the model starts overfitting very early, starting at the 3rd iteration. At that time, the accuracy is still very low, at around 0.7. The best validation accuracy is achieved at epoch 4, but is still very low at 0.73.

In the second experiment, the learning rate for the same model is decayed starting from epoch 4. A slightly better accuracy of 0.75 is achieved. One thing we notice is that in the previous case, the validation accuracy actually became worse after epoch 4, but in this case, the validation accuracy and loss plateaued, but didn’t become worse.



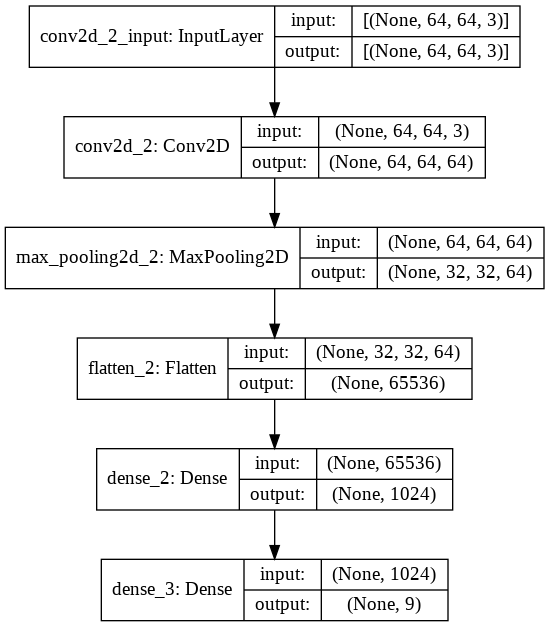
If we compare both these runs as an overlay we find that the decay of the learning rate does have a beneficial effect.





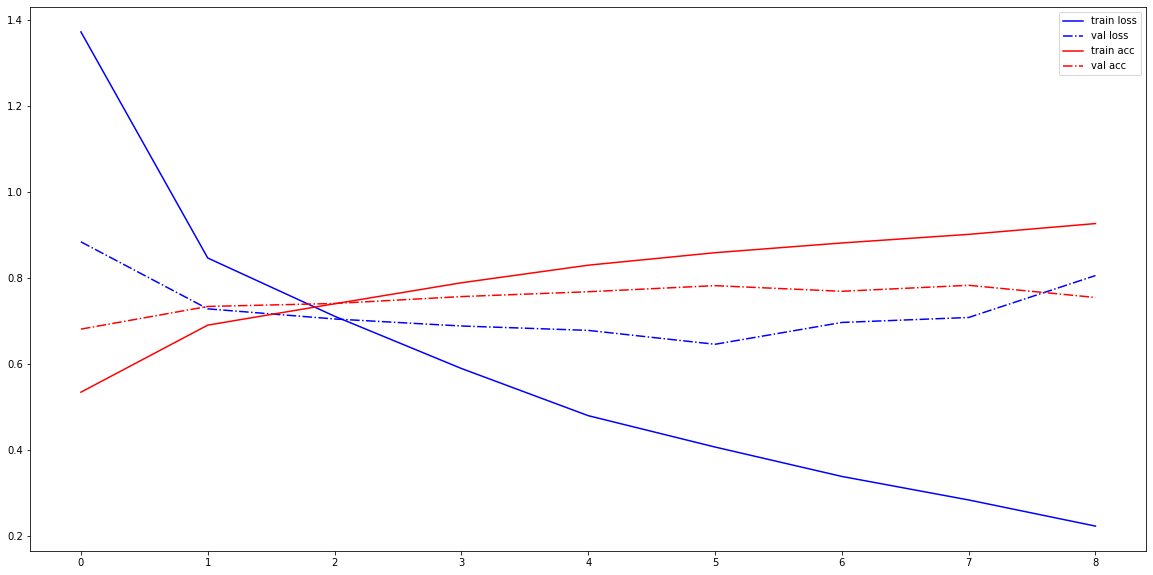
### One Covolution and Two Dense Layers

The network was modified for the next experiment as follows:



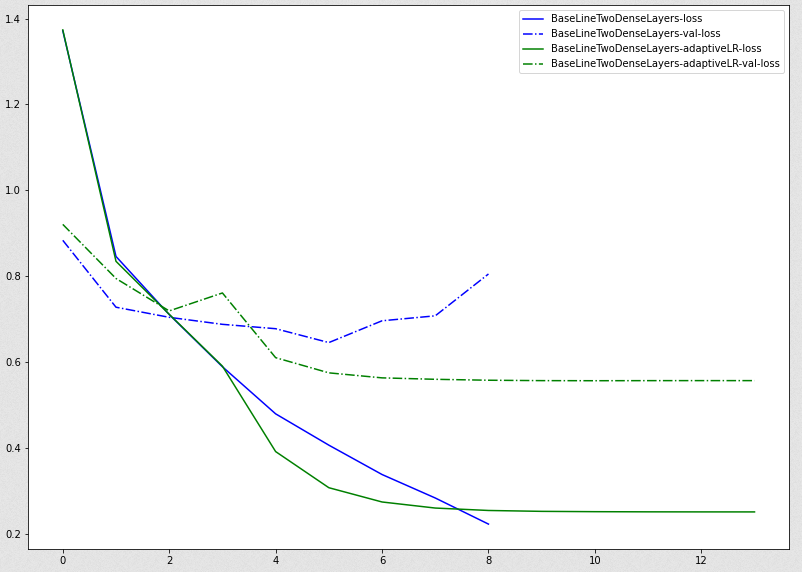
With 2 dense layers, there was an explosion in the number of trainable parameters, which now stood at 67 million!

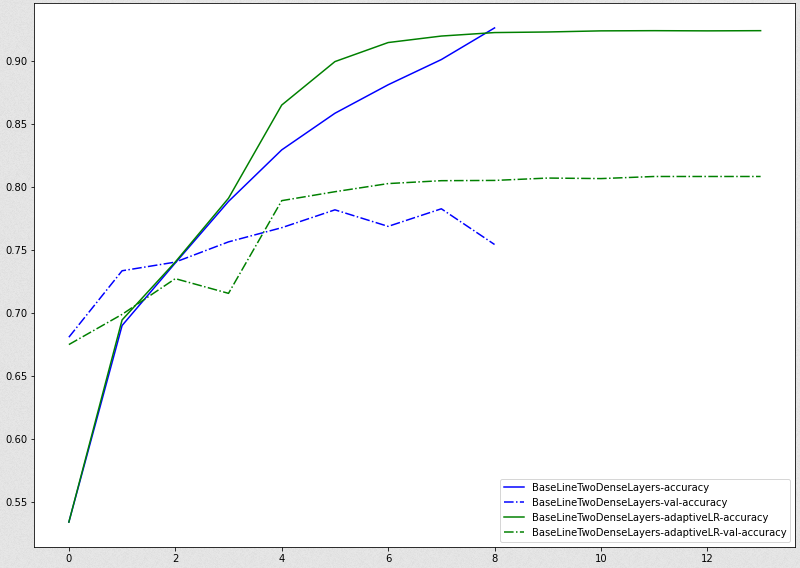
On training this, overfitting could be observed after the 2nd iteration, and grew worse after that. The best validation accuracy was achieved at epoch 7 at 0.78, but at that time, the model had moderate overfitting already.



When the learning rate was decayed, the performance was slightly better, and a validation accuracy of 0.81 was achieved at epoch 12.

Overlaying both the decay and non-decay case, the effect becomes clear.

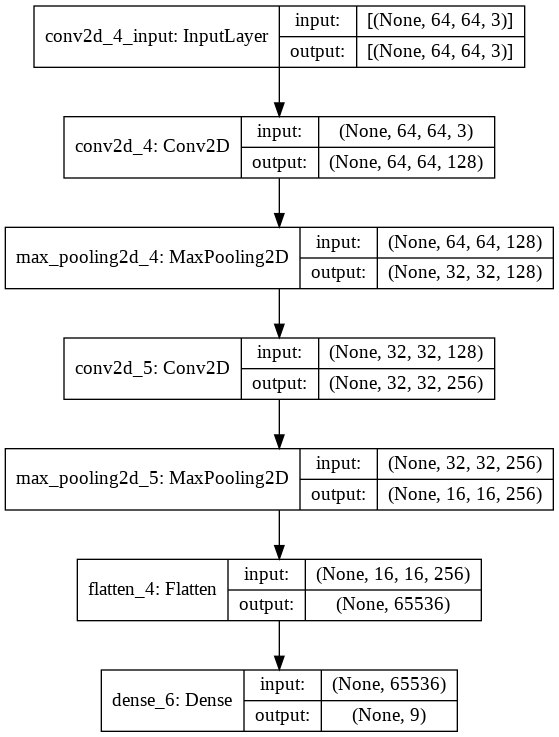




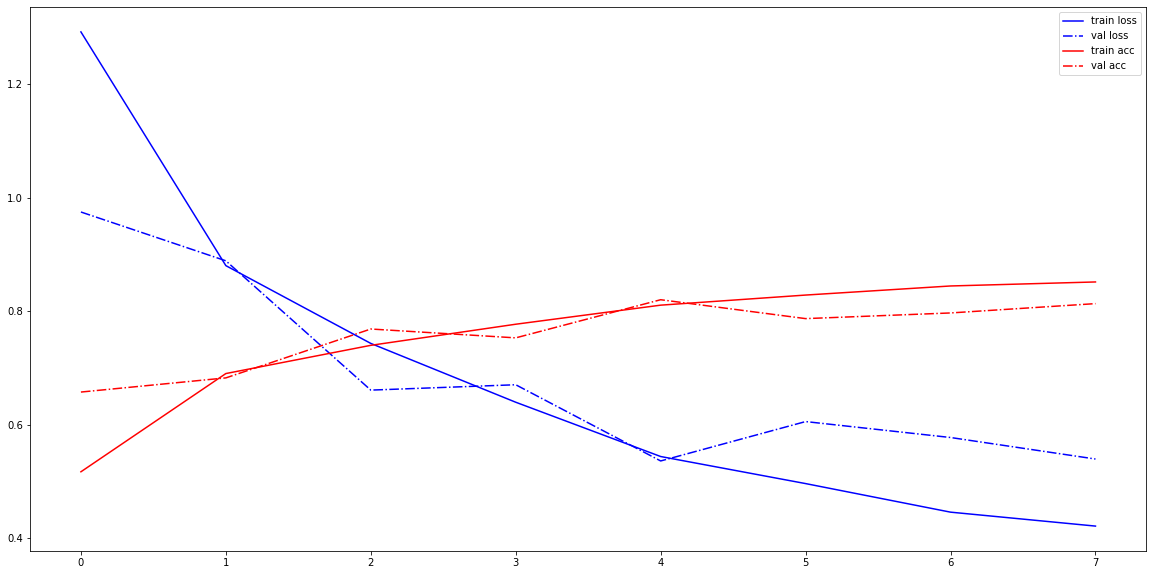
Here we see that while the gap between the training and the validation accuracy increases, both of them follow the same pattern, and plateau out, unlike the previous case where the validation accuracy actually dropped after a while.

### Two Convolutional Layers and One Dense Layer

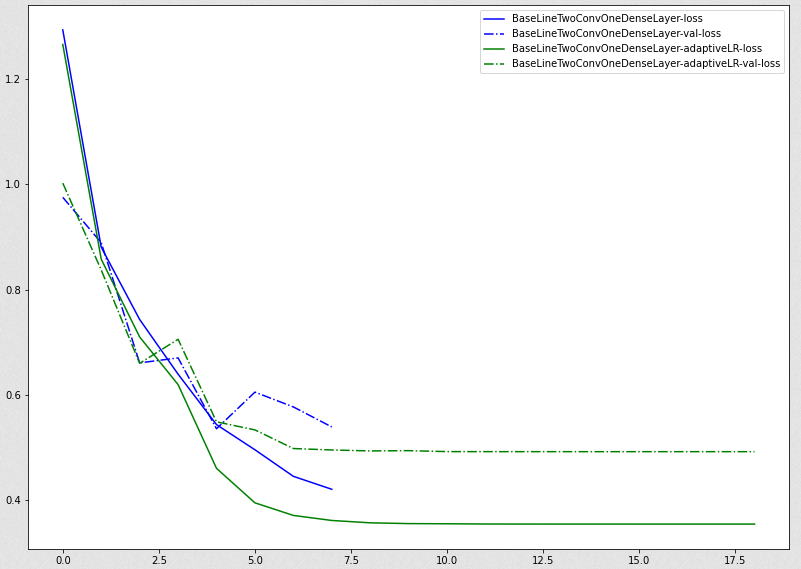
The network was implemented as follows, and the total number of trainable parameters were 0.89 million, significantly less than the previous model.

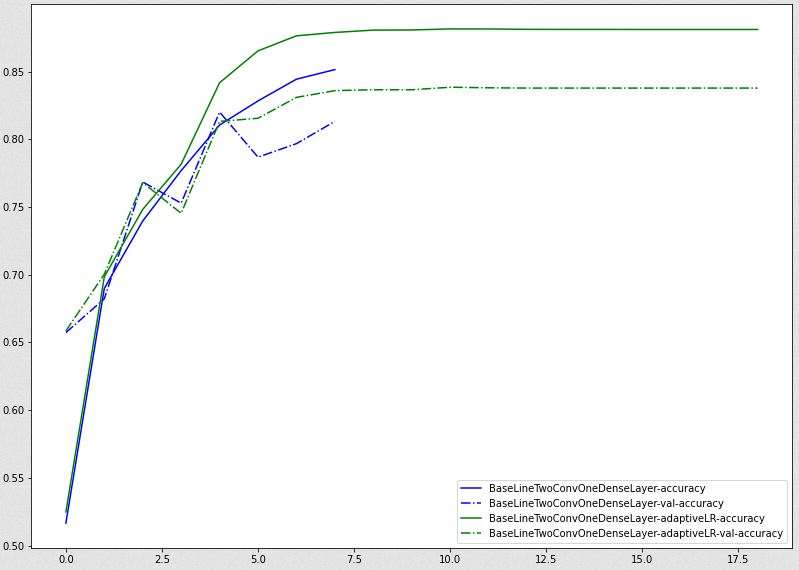


This model achieved a better accuracy of 0.82, and overfitting was significantly reduced.



When the learning rate was decayed, the results improved further and the validation accuracy reached 0.838, and the model was able to train longer and make progress before finally plateauing out.

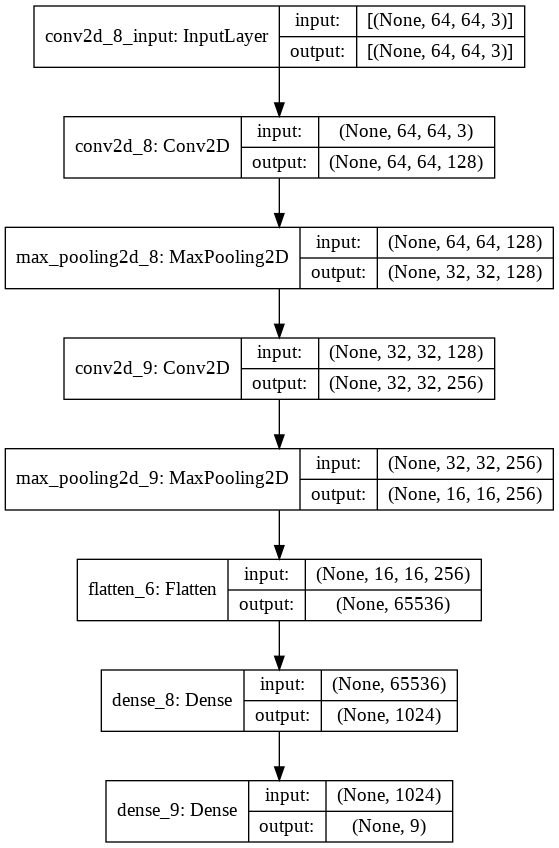




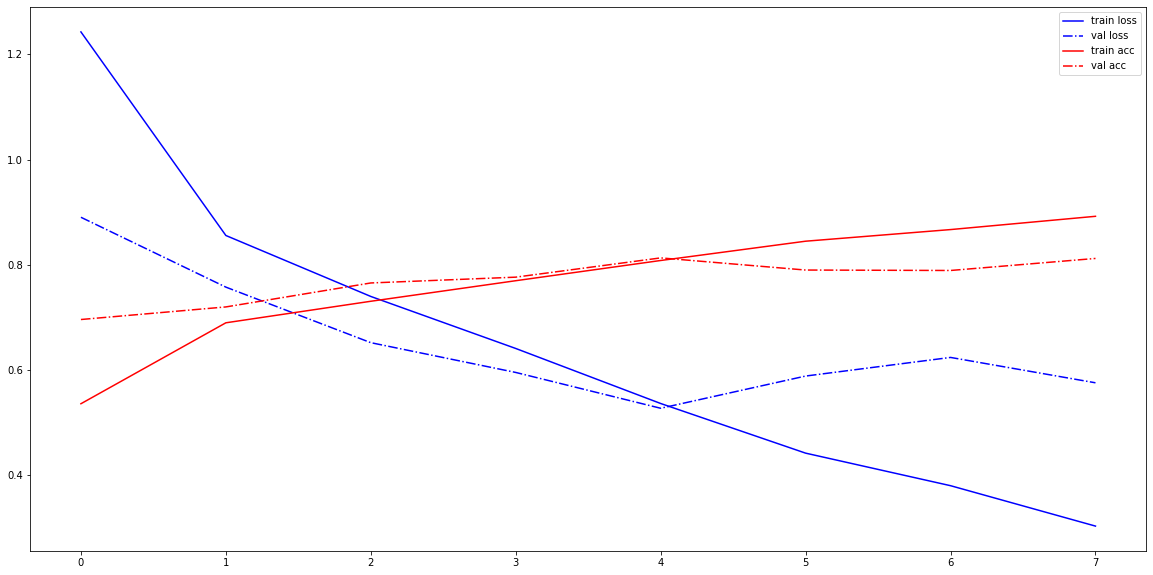
At this stage, increasing the number of convolutional layers has a better effect than increasing the number of dense layers. Also, the number of trainable parameters is kept in check.

### Two convolutional and two dense layers

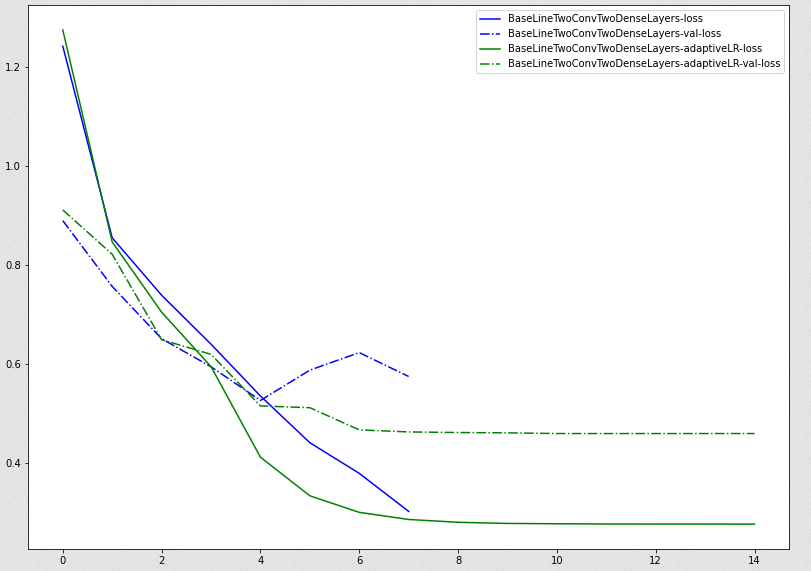
The model was implemented as follows, and the number of trainable parameters were 67 million!

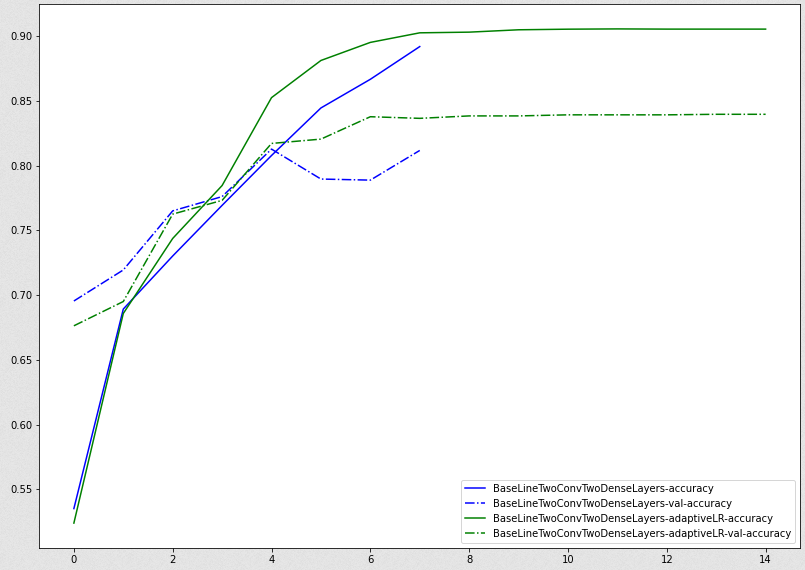


This model performed worse than the model with just one dense layer, and could achieve a best validation accuracy of 0.818 at epoch 4. After that point, there was significant overfitting.



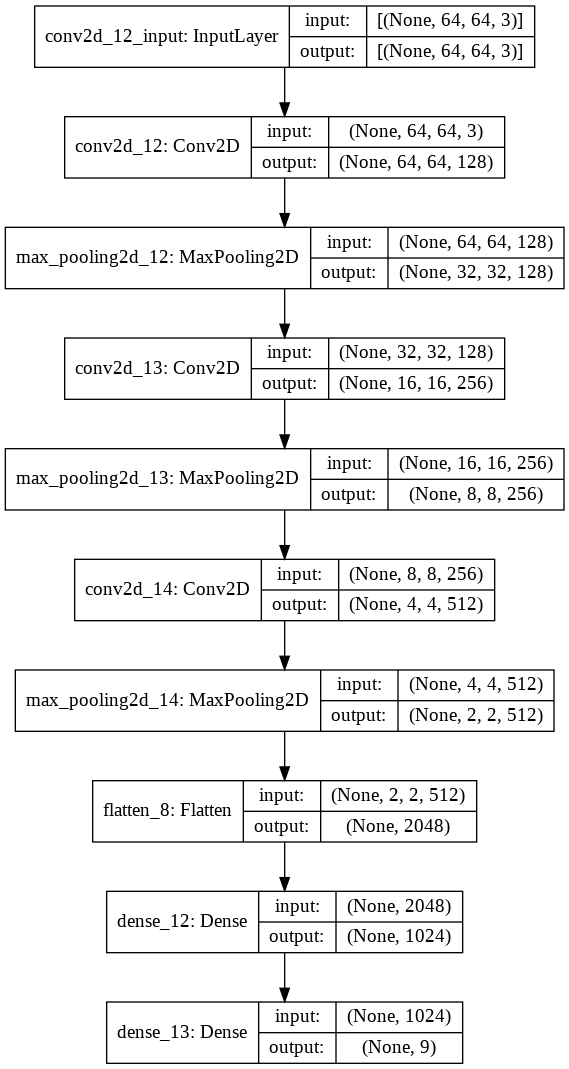
When the learning rate was decayed, the model performed slightly better with the best validation accuracy of 0.84 on epoch 15. Also, while the gap between the train and validation accuracy increased, both followed the same trend, finally plateauing out.



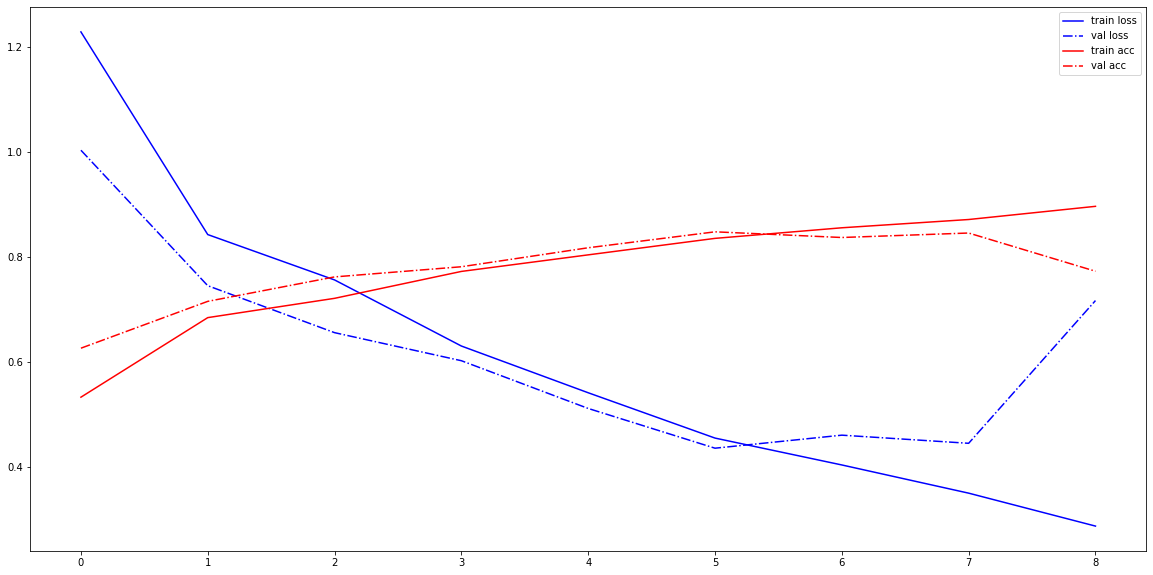


### Three convolutional and two dense layers

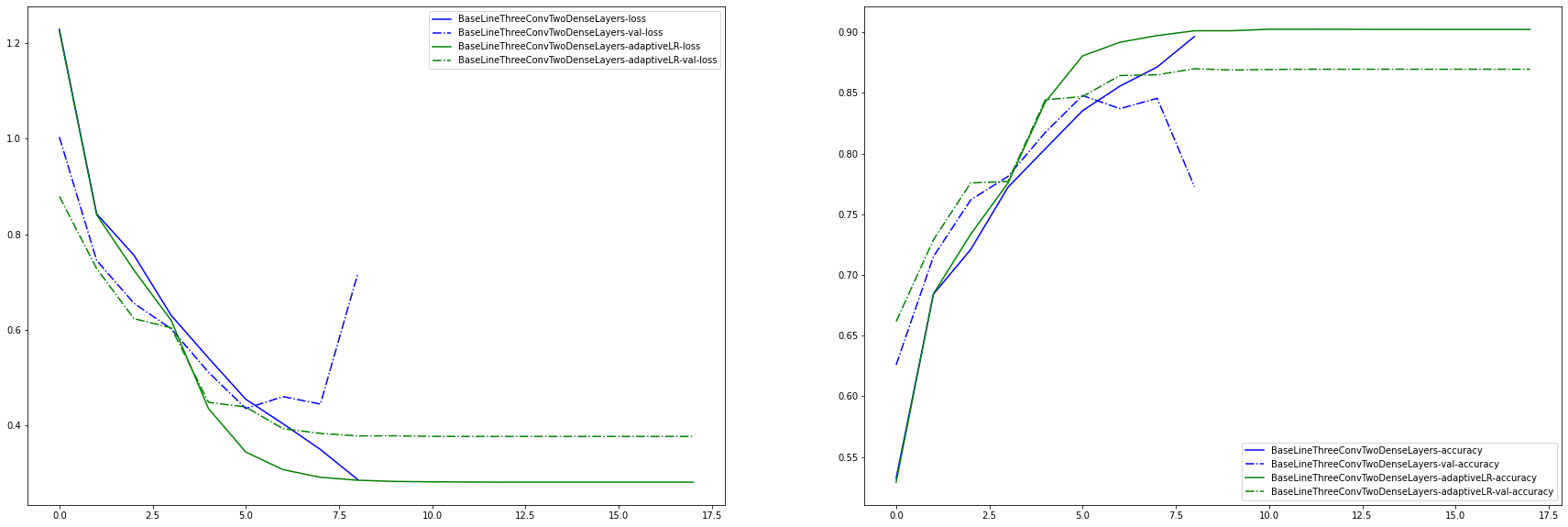
The model was implemented with approximately 3.5 million parameters.



This model performed significantly well compared to the other models, and achieved a best validation accuracy of 0.847 at epoch 5. After epoch 5, the significant overfitting was observed.



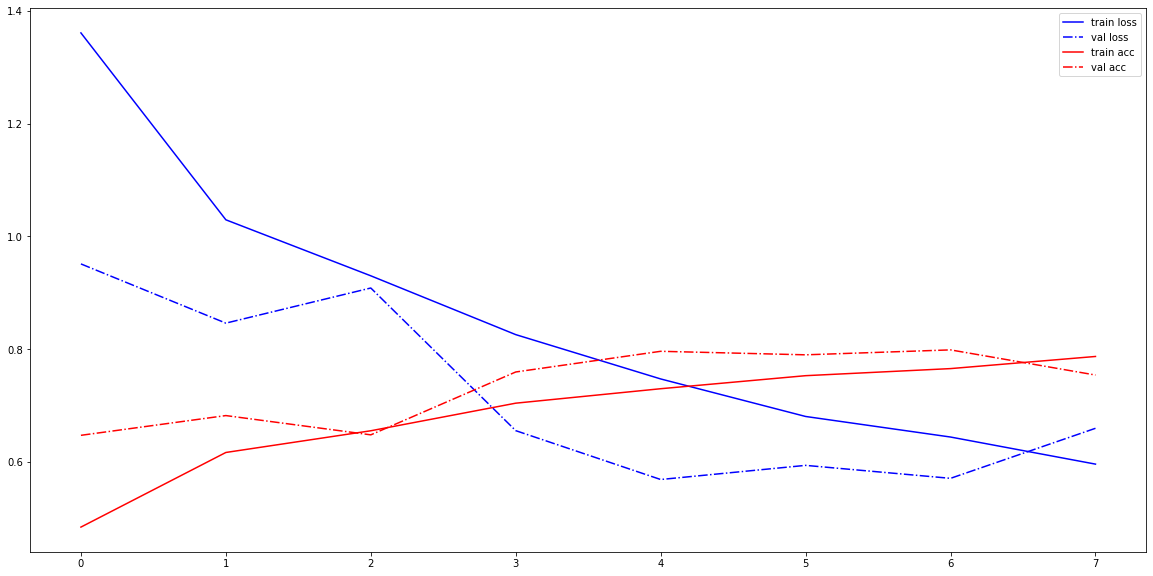
When adaptive learning rate was used, the model performed even better at 0.87. The same patter was observed that the training and test accuracy and losses mirrored each other, although the gap increased.



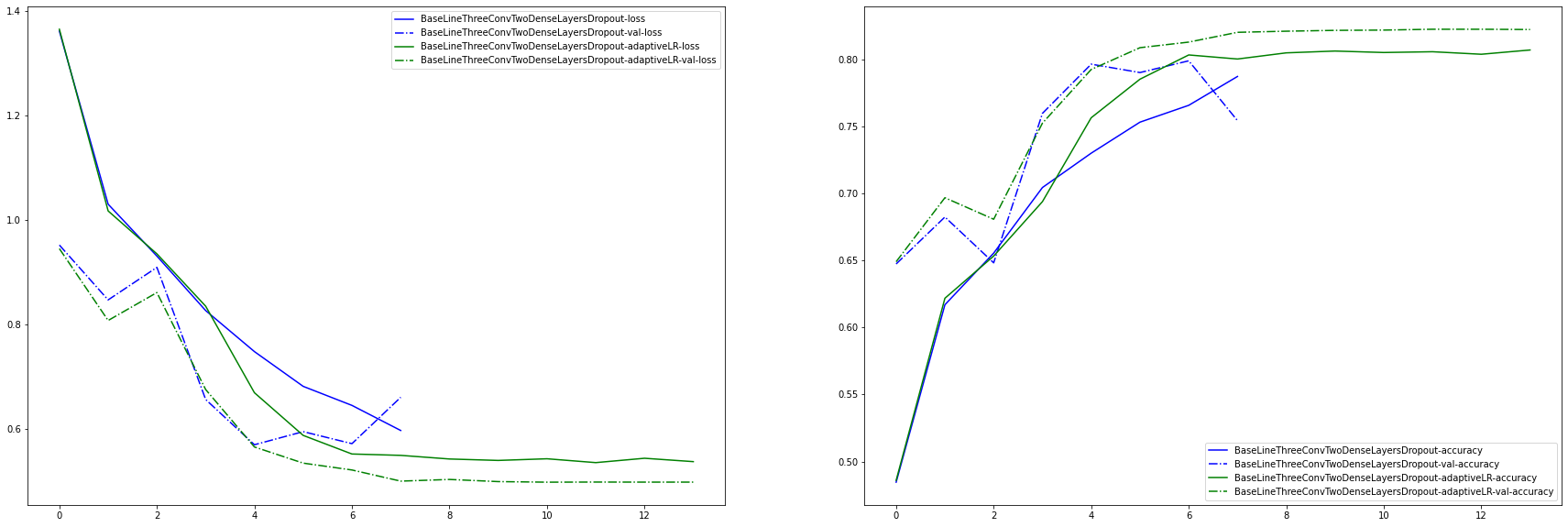
### Three convolutional and two dense layers with dropout

The next experiment was performed to introduce dropout to control the problem of overfitting. The model in the last experiment was adapted to introduce dropout layers (with a probability of 25%). The number of parameters remained the same at 3.5 million.

Introduction of dropout had a benefit in avoiding overfitting for a couple of more epochs, but the validation accuracy actually dropped drastically to 0.798.

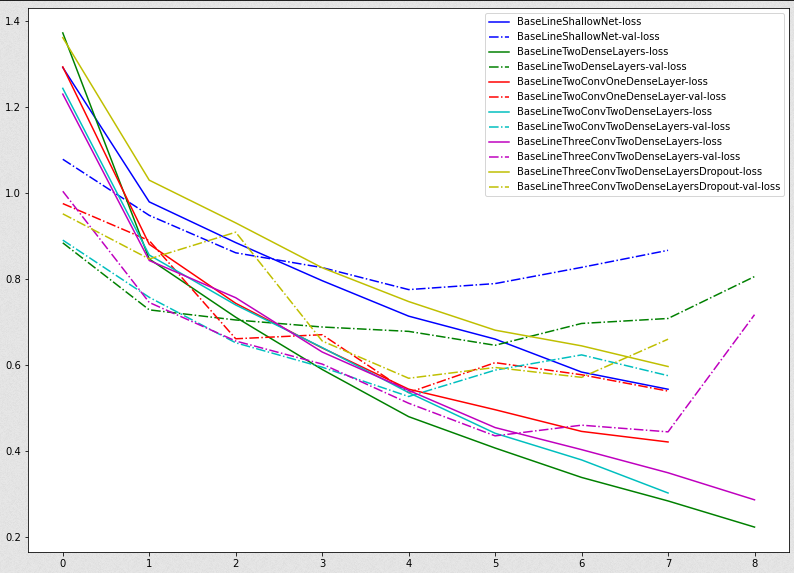


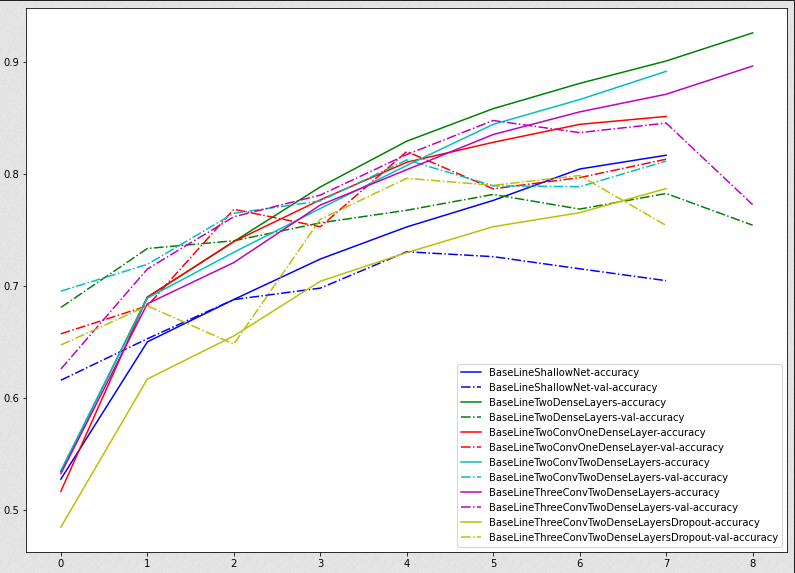
When the learning rate was decayed, the accuracy crept back up to 0.823, but it was still far short of the previous model.



However, one thing we notice here is that dropout had the exact effect that we intended – to reduce overfitting. We see this in the above graph where introduction of dropout results in closing the gap between the train and validation accuracy, much more than any of the previous models we tried out.

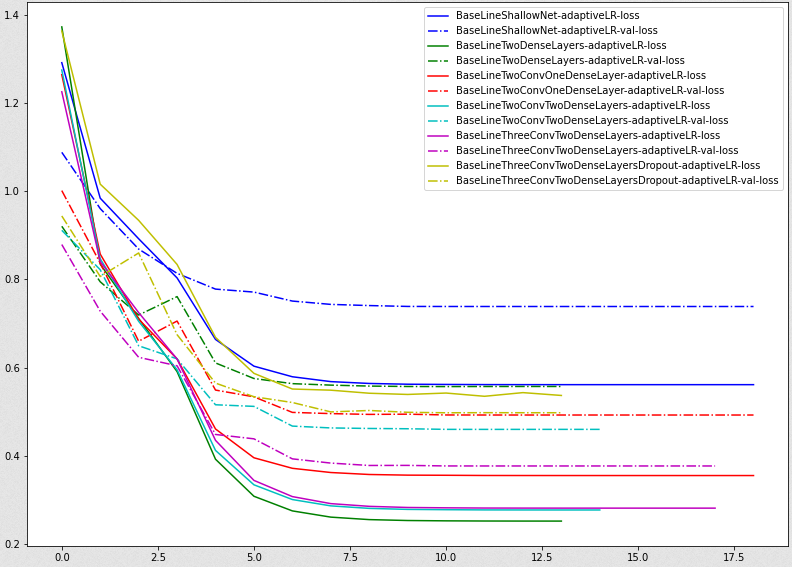
### Comparing all previous runs

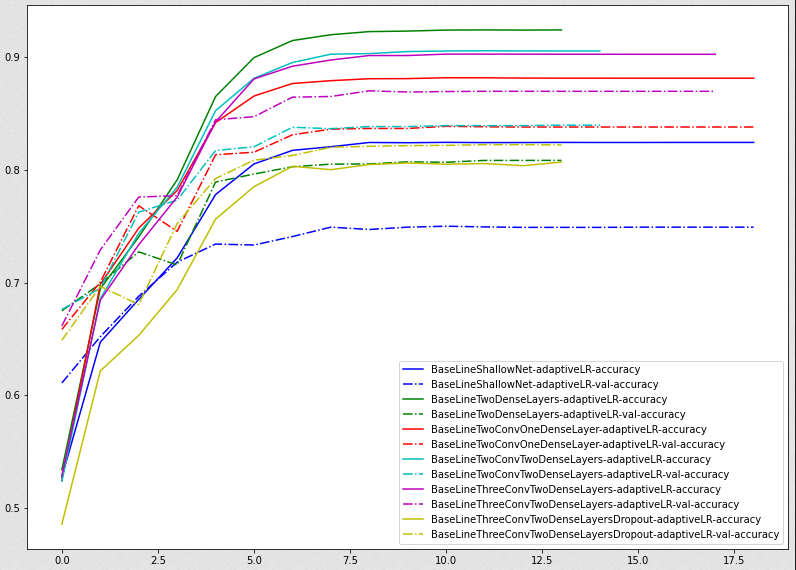




In the above, we see a consistent trend that the validation accuracy of the model increases as the number of convolutional layers increases. Increasing the dense layers doesn’t seem to have as much of an impact as increasing the number of convolutional layers.

With decay of learning rate, we see better performance in validation accuracy and loss:

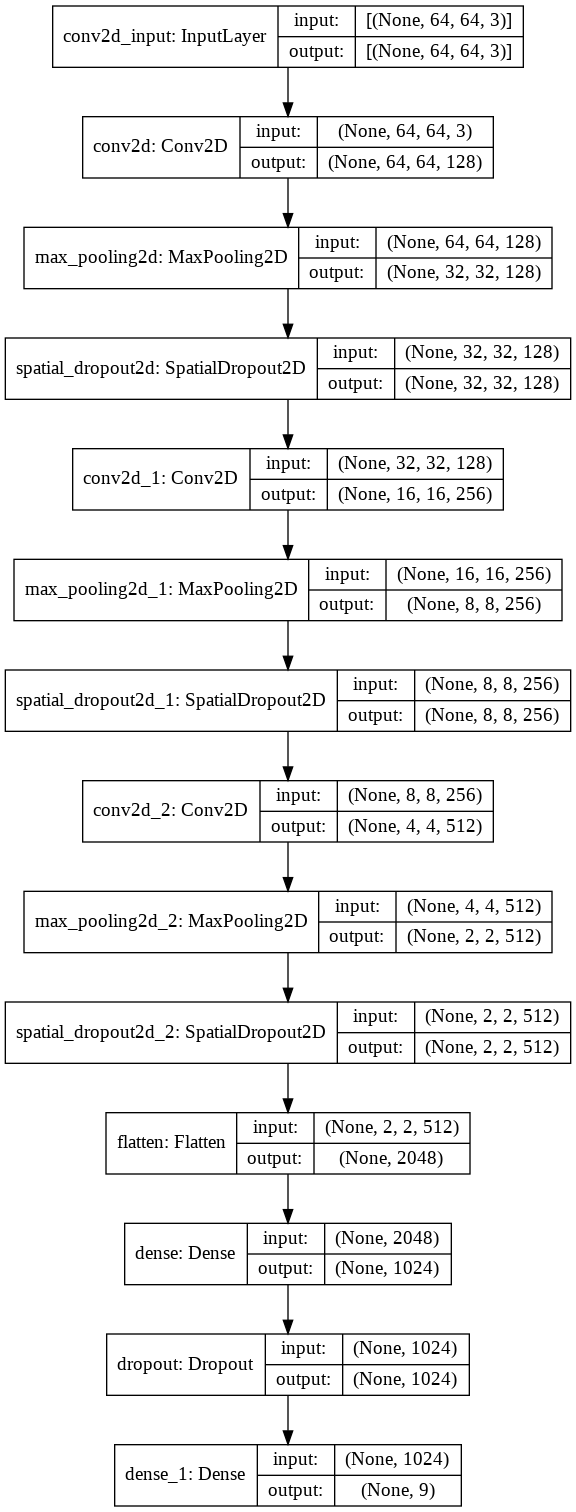




## Data augmentation

### Data augmentation on three convolutional and two dense layers with dropout

The model with three convolutional and two dense layers were chosen for the first experimentation for data augmentation.



The first experiment tried were the following augmentation parameters:

train\_aug **=** tf**.**keras**.**preprocessing**.**image**.**ImageDataGenerator**(**\

rotation\_range**=**20**,**\

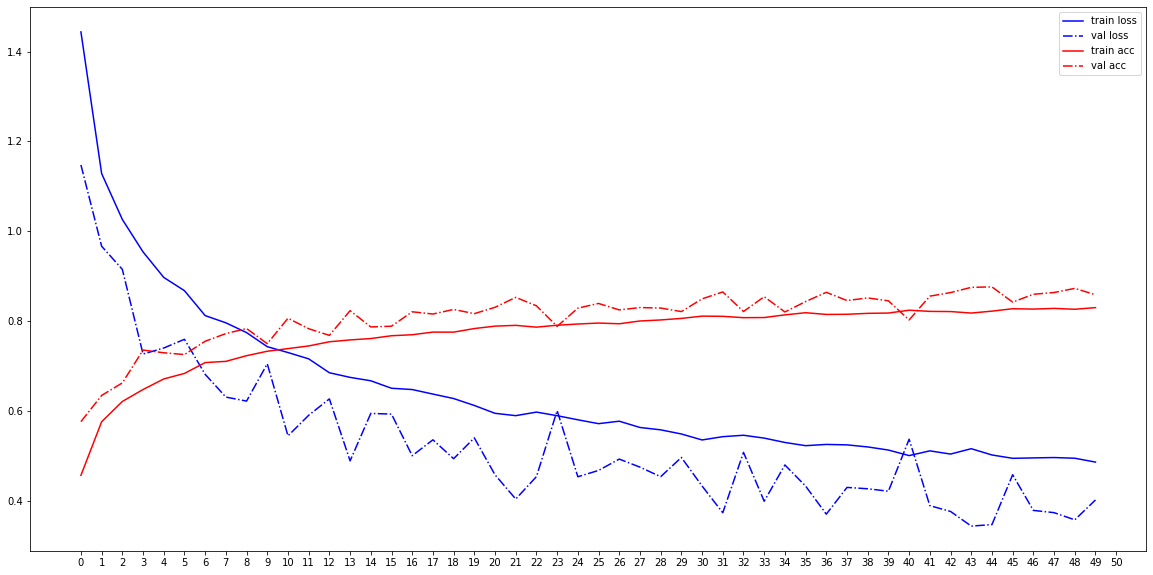
width\_shift\_range**=**0.1**,**\

shear\_range**=**0.2**,**\

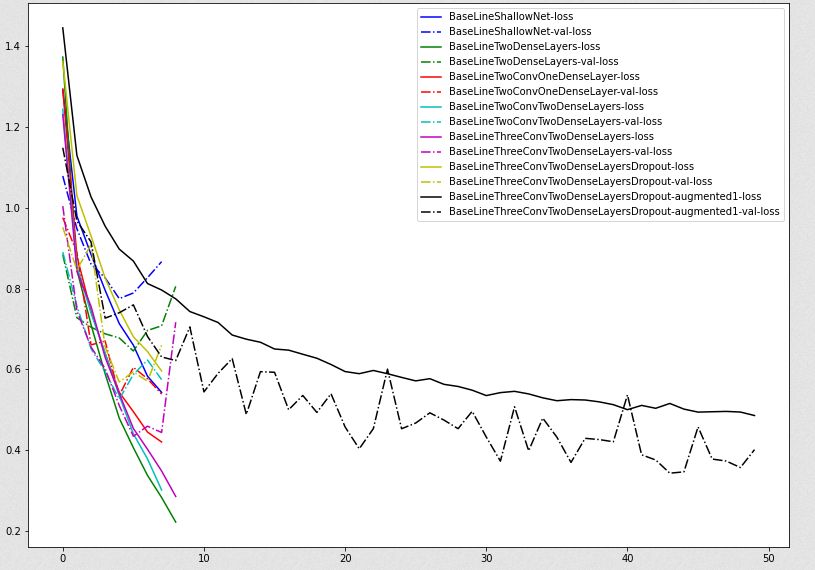
zoom\_range**=**0.4**,**\

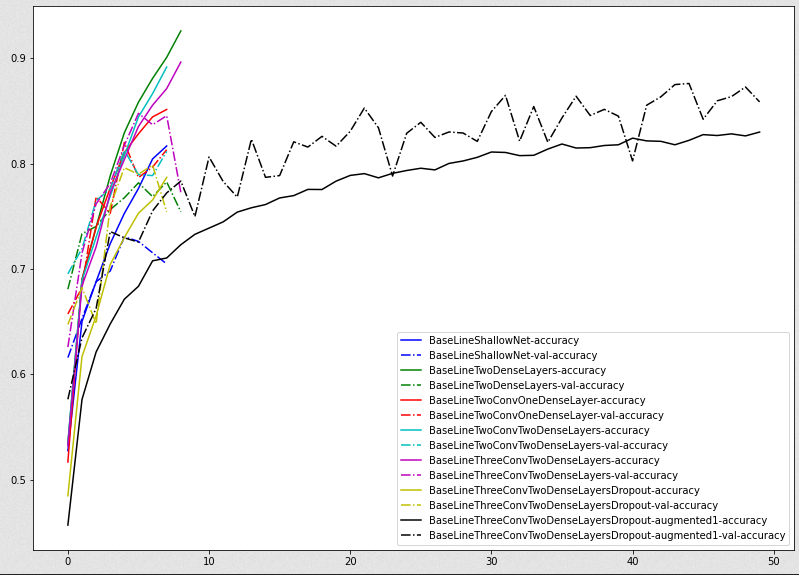
horizontal\_flip**=True)**

We could immediately see that the accuracy become better than any of the previous versions. The best accuracy achieved as 0.876 on epoch 44. This is in stark contrast with the same model without data augmentation, which could only achieve an accuracy of 0.79 and peaked out 4 epochs. We also see that with data augmentation, the model shows no sign of overfitting, although the validation losses and accuracies jump around quite a bit. This is in line with theory as each time it is trained on a different set of data items which introduces this variability.



When the effect of augmentation is compared against the previous methods we get this:





It is interesting to note here that at around epoch 8, the models without augmentation have better accuracy than the model with augmentation, but they start overfitting soon and training has to be stopped; as the augmented model is trained further and further, it beats those models without augmentation.

The second scheme for augmentation that was tried was as follows:

train\_aug2 **=** tf**.**keras**.**preprocessing**.**image**.**ImageDataGenerator**(**\

rotation\_range**=**30**,**\

width\_shift\_range**=**0.1**,**\

height\_shift\_range**=**0.2**,**\

zoom\_range**=**0.4**,**\

vertical\_flip**=**2**,**\

horizontal\_flip**=True)**

This method achieved a slightly lower accuracy of 0.8572 at epoch 39. The training also stopped early at iteration 40, as compared to iteration 44 in the previous instance. The key feature missing in this set of parameters was the lack of shearing, and shearing appears to have some beneficial effect and performs better, although the second instance tried more transforms.

The third scheme tried was the following:

train\_aug2 **=** tf**.**keras**.**preprocessing**.**image**.**ImageDataGenerator**(**\

rotation\_range**=**30**,**\

brightness\_range**=(**0.80**,** 1.2**,),**\

width\_shift\_range**=**0.1**,**\

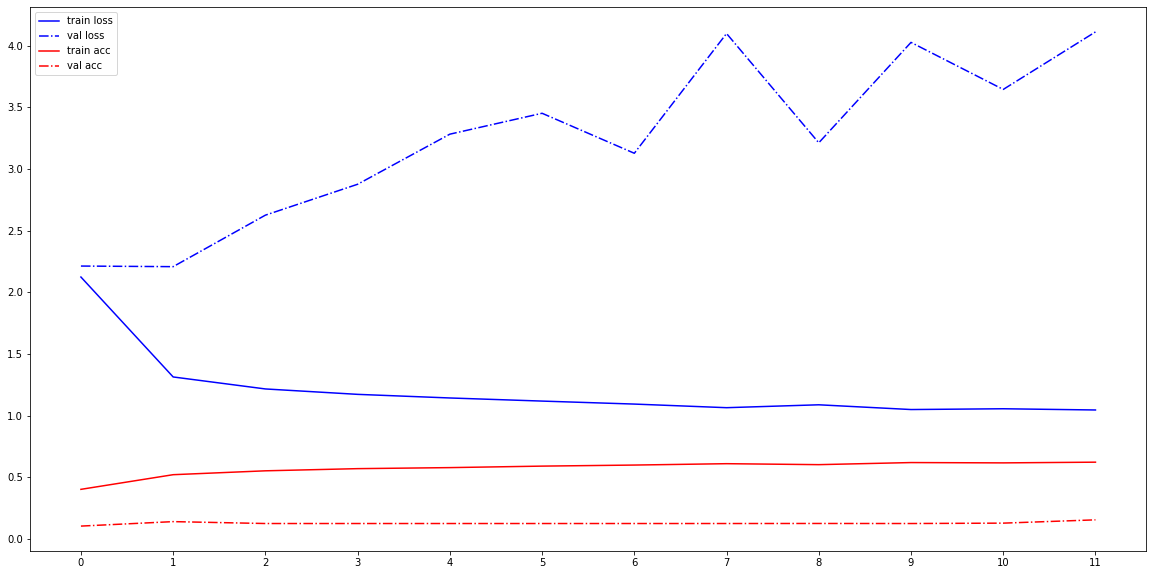
height\_shift\_range**=**0.2**,**\

zoom\_range**=**0.4**,**\

vertical\_flip**=**2**,**\

horizontal\_flip**=True)**

This model performed really bad and never converged. The best accuracy it could achieve was 0.15



Further experimentation revealed that specifying the brightness\_range was the reason the accuracy dropped so low, and no combination of values for this parameter could be found which would result in good performance.

The last augmentation scheme tried was the following:

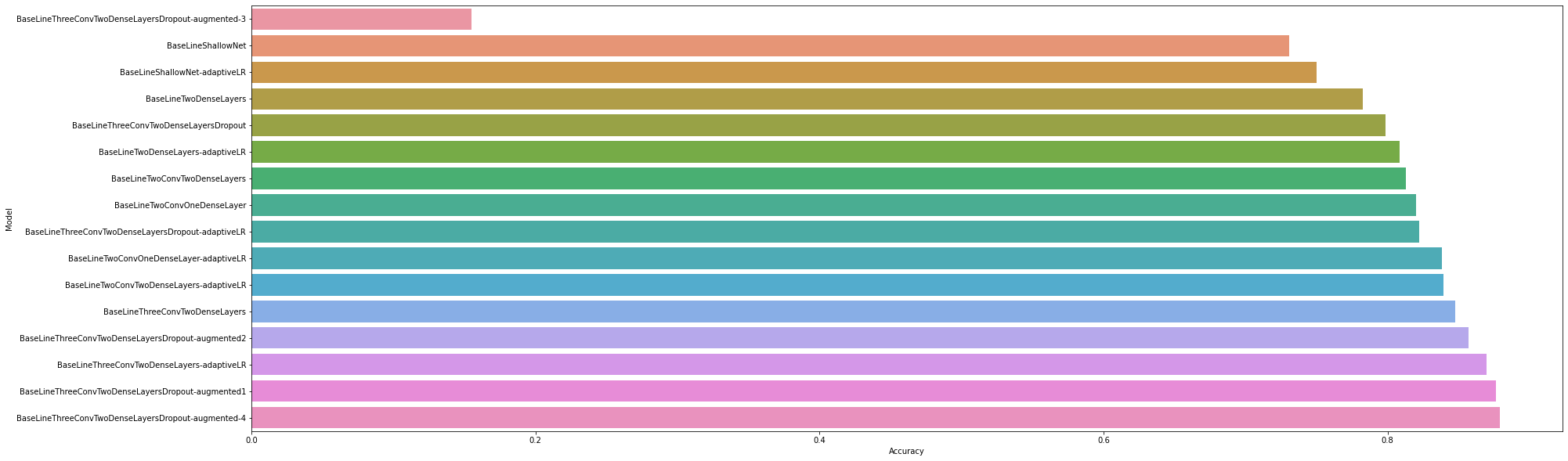
train\_aug **=** tf**.**keras**.**preprocessing**.**image**.**ImageDataGenerator**(**\

rotation\_range**=**20**,**\

horizontal\_flip**=True)**

This was able to achieve an accuracy of 0.879 at epoch 33.

When we compare all the models, we get the following accuracies.



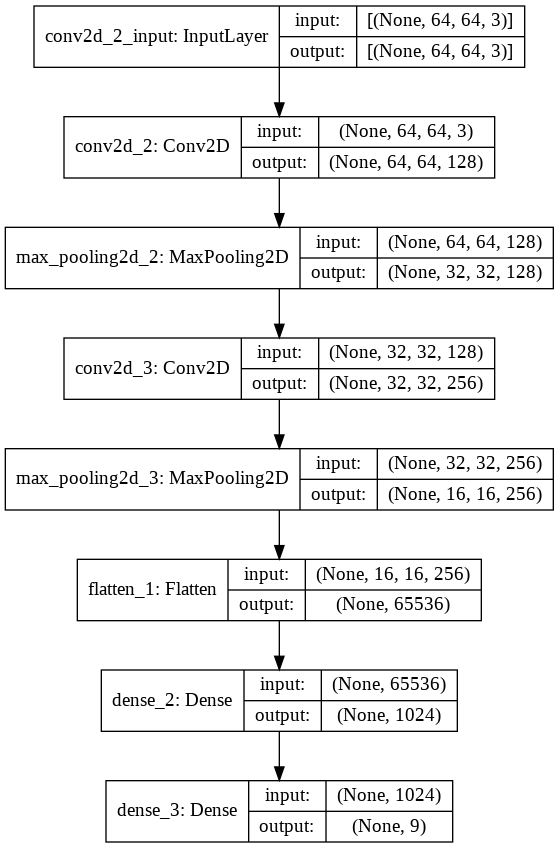
Here we see that data augmentation improves the performance the most. However, the model with three convolutional and two dense layers without dropout and decaying learning rate also achieves a good result.

With respect to data augmentation we observe the following:

1. We see no consistent rule that we can apply, and which transforms have the best results must be empirically established.
2. We see that introducing a shear transform has a beneficial effect in this dataset
3. We see that changing the brightness level leads to a much decreased accuracy and the model never converges. This could suggest that the values of the pixels matter in a CNN, and not just the gradients and differences in levels between the pixels.

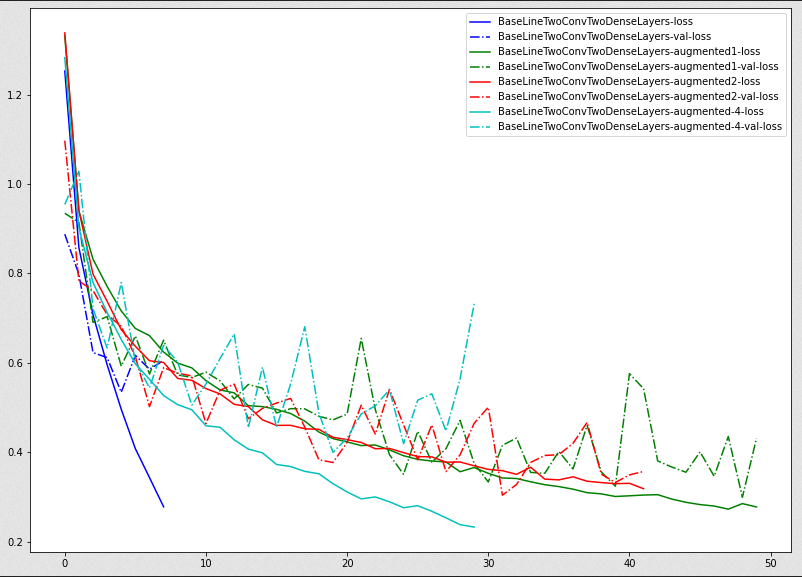
### Data augmentation on two convolutional and two dense layers

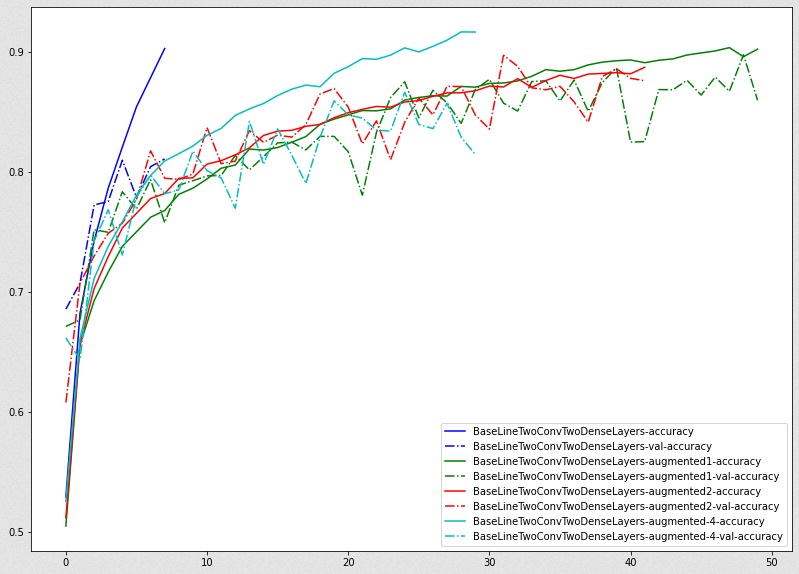
A second model was tried for data augmentation that had two dense and two convolutional layers, and no dropout layers. The number of trainable parameters were significantly lower than the previous model.



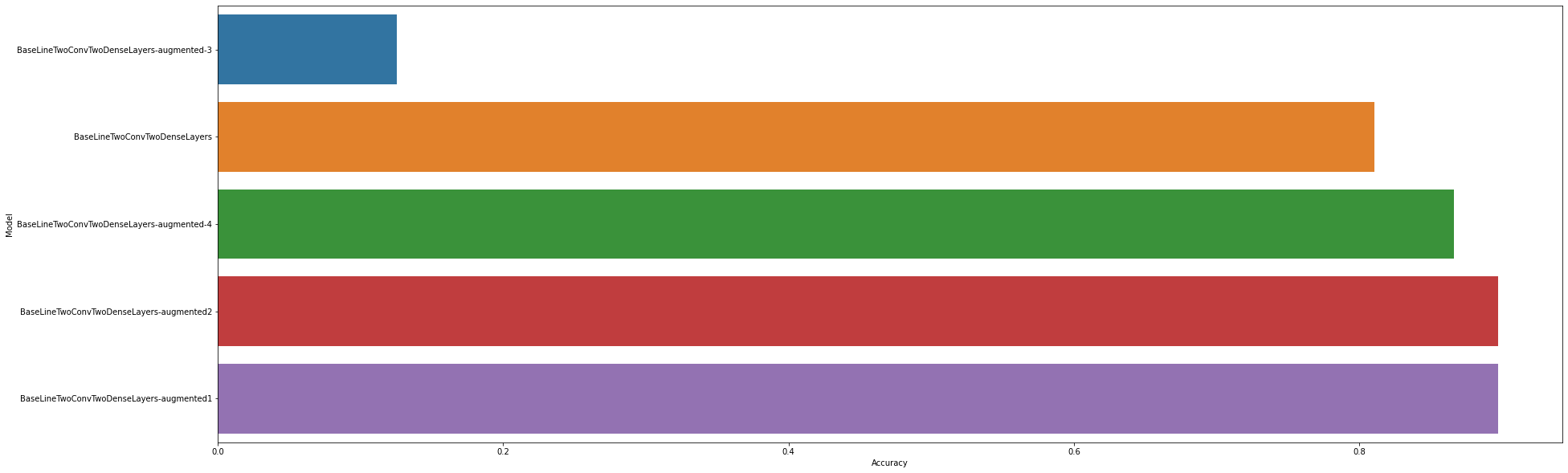
The same set of parameters were tried for data augmentation.

Again, with this model as well, the run where the brightness range was specified performed very badly and failed to converge. This run has been left out of the report because the results are similar and it makes it more difficult to see the results of the other runs.





The best accuracy of 0.8975 was achieved the first augmentation scheme. Note that in the previous model, the best accuracy was achieved by the 4th augmentation scheme, but was lower than this (0.8794).

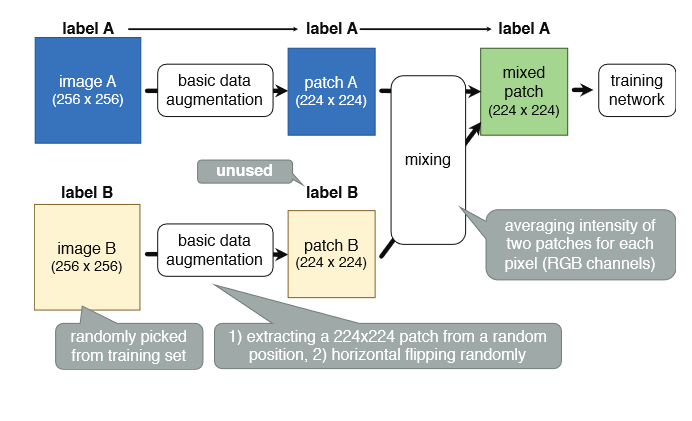


The general take aways are these:

1. In general adding convolutional layers has a greater effect than adding dense layers
2. The key lies in training as many epochs as possible while avoiding overfitting
3. A smaller model with fewer parameters may perform better than a larger model with more parameters
4. Data augmentation has a bigger effect on the validation accuracy than dropout. This is expected as while dropout only makes weights zero, data augmentation actually gives the network more opportunity to learn.
5. Different augmentation schemes perform differently on different networks, and the best augmentations scheme must be empirically determined.

In general data augmentation would be most effective when the set of transformations are representative of the variety found in actual images that the model will need to predict.

## Other Methods of Data Augmentation

1. Mixing Images: two images are mixed together by taking the average intensity of each pixel.   
     
   Inoue[[1]](#footnote-1) report an improvement in classification accuracy by using this method, although it may be counter intuitive.
2. Randomly erase patches within an image [[2]](#footnote-2)
3. Use a GAN or a VAE to generate artificial instances
4. Instead of generating new images, we can generate an intermediate representation. The training network will then have to be split into two, a part that produces the intermediate representation, and a part that classifies them. A VAE or GAN can be used to conjure more instances of the intermediate representation  
   Since the intermediate representation has a lower dimensionality than the original images, they may be easier to generate.
5. Injecting noise into the image
6. Neural style transfer to generate new instances[[3]](#footnote-3)

# Few Shot Learning

Few shot describes the set of learning tasks where there are d classes, and one or more of those classes have very few training examples. One-shot learning is a special case of few-shot learning where one or more classes have just one training example. Zero-shot learning is a special case where one or more classes have no training example.

## One shot learning

A familiar task for one-shot learning is face recognition software for access and verification. Typically, we have just a few photographs of every person available, and given a face, we should be able to verify the face. The small number of training examples is not enough for any machine learning model to learn to recognize the face. Furthermore, even if we had enough number of examples of every face, every time a new face was added, it would require training the entire model from scratch again using all the data, and that is not a viable option. Deep Neural Networks are also prone to catastrophic forgetting, and hence incremental training with just newly added data may not be possible.

One way to get around these problems is to learn a *similarity function* for faces. Given two faces, the function outputs the measure of similarity between the two faces. This function can be learnt from training data where many images of a person’s face is present. Once the model is trained and the function is found, new faces can easily be added without retraining, and the function will still give the similarity score for new faces.

One approach used here is to train neural networks to output embeddings or vector descriptors of each face. This maps the face onto a lower-dimensional vector space. A Siamese network is used on a pair of images, and two images, and two vector maps produced. The distance between the vector maps can be used to predict whether the faces are same or not. The Siamese-network is the same neural network with the same set of parameters and weight.

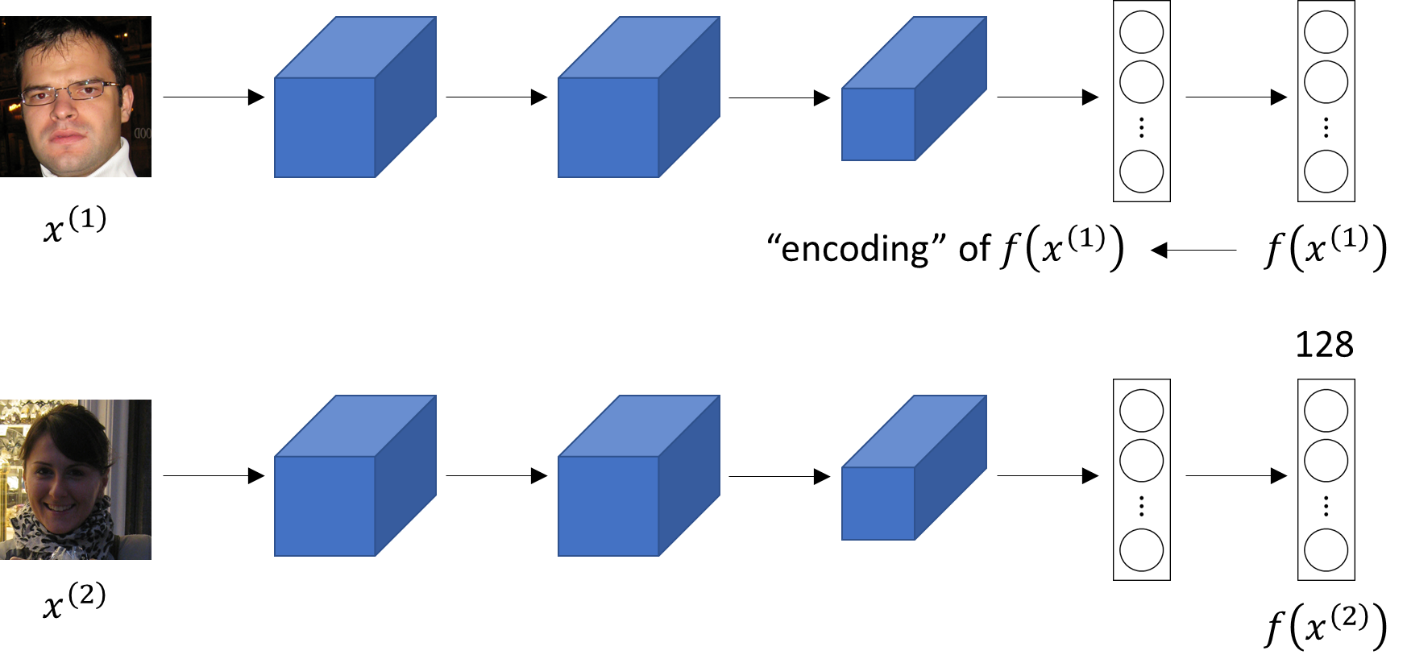
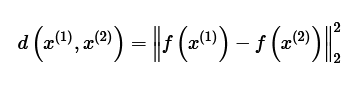


Figure 1Source http://datahacker.rs/one-shot-learning-with-siamese-neural-network/

The loss is given by



And d must be small for two faces of the same person, but large for two faces of different persons.

An improvement here is to use the Triplet loss function. In this method, we look at three faces, the anchor face (A), the positive face (P) and the negative face (N).

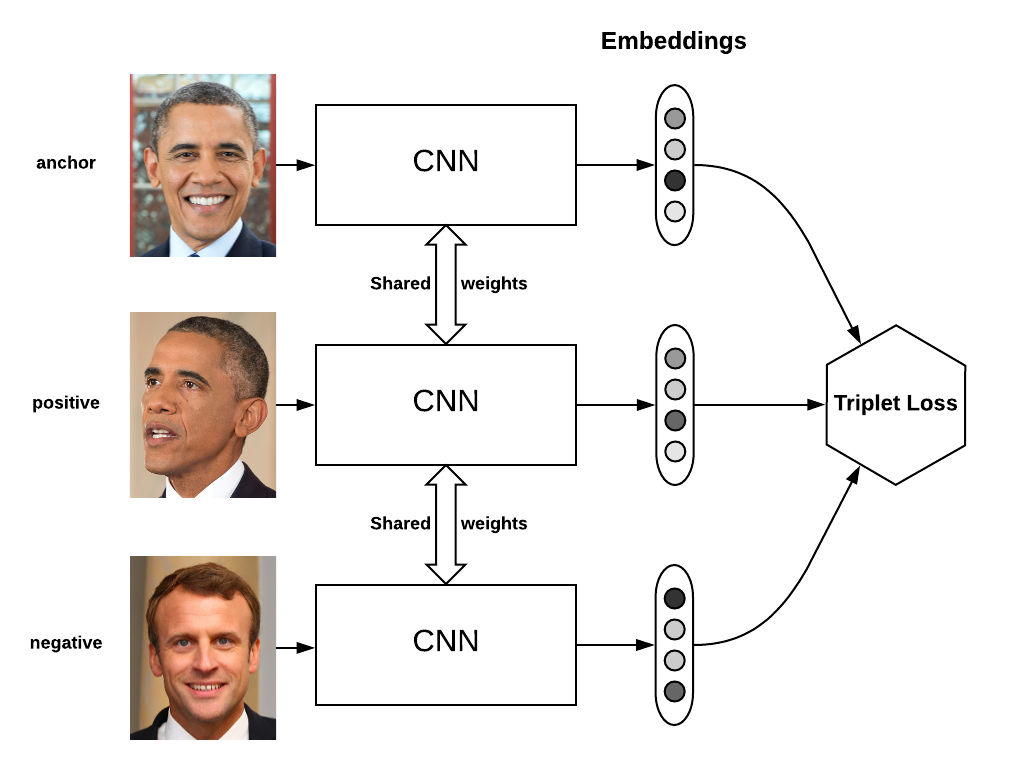


Figure 2 Source <https://omoindrot.github.io/triplet-loss>

The loss function is modified as follows:



During training time, we do need multiple images of the same person, but at test time, we only need one image of a person. Also, to add a new face to the database, we don’t need to retrain the network with multiple images, and the function learns how to tell that two images are the same.

One small point to note here is that the choice of triplets during training is important. If we choose randomly, then the faces are unlikely to be of the same person, and since they are different they will be very easy to distinguish. Therefore the network will not learn anything as it will always predict the correct output very easily. The key here is to take faces that are hard to tell apart and learn on them.

One example of such a network is DeepFace[[4]](#footnote-4).

Another way to approach the same problem is to treat face recognition as a logistic regression problem. This is done in a two-step manner

1. Learn a DNN to find the encoding for a face
2. Given two encodings for two faces, find the difference between the encodings, and use a logistic-regression (or other ML model) to output whether the faces are same or not.

This converts the task of face verification to a binary classification process.

## Zero Shot Learning

Zero-shot learning is another extreme of few-shot learning where we may not have any training instances of some classes. Zero-shot learning was first spurred by attempts to read the mind from fMRI images. Since such images have very few training examples, it is difficult to train a machine learning algorithm.

The key here, again is to convert every training example into a multi-dimensional vector embedding. For example a vector embedding of a face could include (color of eyes, length of nose, width of face, length of face, color of hair, …). These are called *category vector embedding*.

The next step is to learn a DNN (or other machine learning model) to map from the feature space to the category vector embedding.

At training time, we need a lot of embeddings to learn the function that can perform the mapping.

However, at test time, we can use the same function that we learnt already to create a map for an unseen example. The unseen example may belong to a category for which we have not seen any training example so far.

There are two ways to approach the problem:

1. For any instance at test time, find the category vector encoding v, and use nearest neighbor clustering to determine its class. Note here that we need to know the embedding of the class for which there are no training examples, but we don’t need any training examples for that class. However, the drawback of this simple approach is that there is no guarantee of separability.
2. The second approach is to use the energy function. During training we learn an energy function , such that this is large when the data and task match, and small when it doesn’t match. A common choice is a bilinear function. At test time, we just need to get the maximum value possible for this function. The advantage of this approach is better separability.[[5]](#footnote-5)

One example of approach 1 is by McCartney[[6]](#footnote-6). Xie and Virtanen[[7]](#footnote-7) use approach 2 for audio classification.

## Theoretical underpinning of few-shot learning

Wang et al. [[8]](#footnote-8) provide a good discussion on the theoretical underpinnings of few-shot learning. One way to approach any machine learning task is to see it as an empirical risk minimization problem. Specifically we want to minimize the following:



Where p(x, y) is the true probability of (x, y). Since this true probability is not known, the best we can do is to minimize the total loss over all the instances.

We try to estimate a hypothesis h over a hypothesis space H which can lead to the least total loss. The notation used here is as follows

1. is the function that minimizes the expected risk . is the true function that minimizes the objective
2. is a function in H that minimizes the expected risk. This is a projection of on the hypothesis and is the best possible approximation of .
3. is a function in H that minimizes the empirical risk. This is the function is what is learnt by the model

Wang et al. explain this diagrammatically:

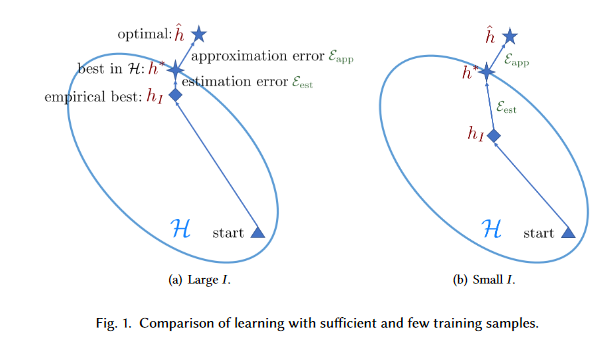


Figure 3 Wang et al. Generalizing from a few eamples: a survey on few-shot Learning

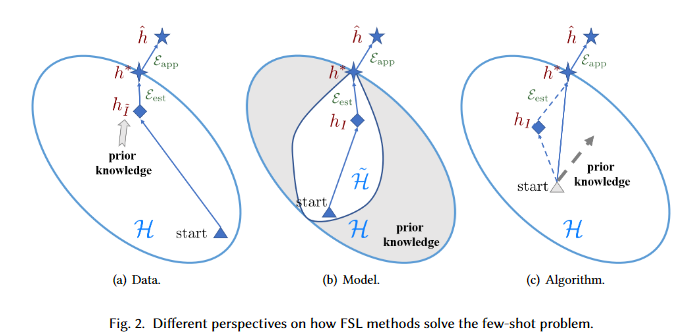


Figure 4Wang et al. Generalizing from a few eamples: a survey on few-shot Learning

The problem here with few shot learning is that our starting point may be very far away from a good , and because there are very few examples, we may not be able to learn a good .

The problem is approached in 3 ways:

1. Data – by providing more data by augmentation or some other methods to generate data from prior knowledge
2. Model – the hypothesis space is reduced by using prior knowledge, and now the search is in a smaller space
3. Algorithm – to use a heuristic to choose a better starting point in the hypothesis space

They provide the following taxonomy:

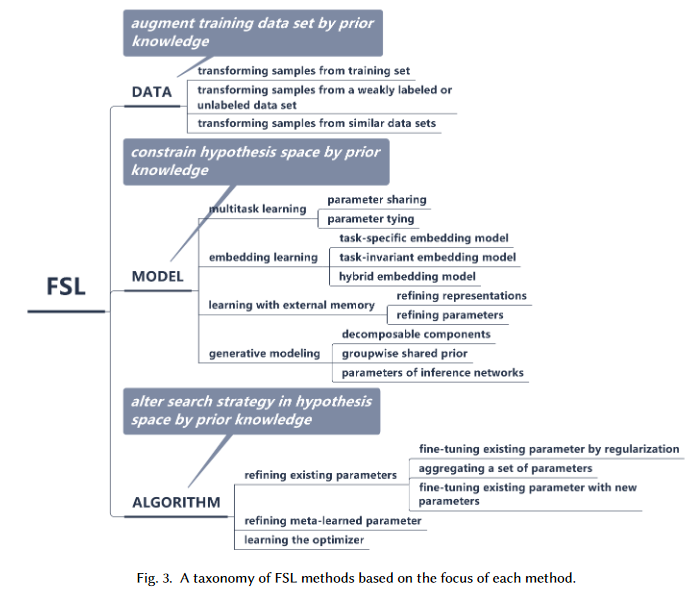


Figure 5Wang et al. Generalizing from a Few Examples: A Survey on Few-shot Learning

# Ensembles

For this part of the assignment, ten different neural networks were trained on the data, and their output predictions were then aggregated to form the ensemble predictions.

Two different schemes for aggregating the predictions were used:

1. The individual probabilities output by the softmax layers for each class were summed up and the maximum value taken for the final prediction of the class
2. All the ten different predictions, and the sum of all the probabilities for each class were combined to form a new feature vector, and this was passed to a secondary classifier for the final prediction.

For the source of variability of the base learners, a model generator was created. The model generator varied the following:

1. Stride size
2. Filter shape
3. Whether to use pooling at each layer or not
4. Size of the pooling
5. Number of convolutional layers
6. Number of dense layers
7. Number of neurons in each dense layer
8. Activation function of each dense layer (except the output layer)

To prevent the models from growing too large to be reasonably trained, the above values were selected from a pre-determined range of values which were not infinite.

A model factory class was created as below:

**class** **ModelFactory:**

*@staticmethod*

**def** get\_conv\_layer**(**depth**,** inshape**=None,** force\_stride\_size**=False):**

out\_layers **=** **[]**

filter\_size **=** random**.**choice**([**2**,** 3**,** 4**,** 5**])**

stride\_size **=** random**.**choice**([**1**,** 2**,** 3**])**

stride\_size **=** stride\_size **if** stride\_size **<** filter\_size **else** filter\_size

# In case there is an exception, the caller will

# set this and call again

stride\_size **=** 1 **if** force\_stride\_size **else** stride\_size

stride\_size **=** **(**stride\_size**,** stride\_size**,** **)**

filter\_shape **=** tuple**([**filter\_size**,** filter\_size**,])**

act\_fn **=** random**.**choice**([**'tanh'**,** 'relu'**,** 'sigmoid'**,** 'selu'**,** 'elu'**])**

pd **=** random**.**choice**([**'same'**,** 'valid'**])**

**if** **None** **!=** inshape**:**

conv\_layer **=** tf**.**keras**.**layers**.**Conv2D**(**\

depth**,**\

filter\_shape**,**\

padding**=**pd**,**\

input\_shape**=**inshape**,**\

strides**=**stride\_size**,**\

activation**=**act\_fn**)**

**else:**

conv\_layer **=** tf**.**keras**.**layers**.**Conv2D**(**\

depth**,**\

filter\_shape**,**\

padding**=**pd**,**\

strides**=**stride\_size**,**\

activation**=**act\_fn**)**

out\_layers**.**append**(**conv\_layer**)**

**if** random**.**choice**([True,** **False]):**

pool\_size **=** random**.**choice**([**2**,** 3**,** 4**])**

out\_layers**.**append**(**tf**.**keras**.**layers**.**MaxPooling2D**(**\

pool\_size**,**\

pool\_size**,))**

**if** random**.**choice**([True,** **False]):**

out\_layers**.**append**(**tf**.**keras**.**layers**.**SpatialDropout2D**(**0.25**))**

**return** out\_layers

*@staticmethod*

**def** generate\_model\_internal**(**inshp**,** nlabels**:**int**,** modelname**:**str**,** fss**:**bool**):**

layer\_neurons **=** **[]**

n\_conv\_layers **=** random**.**choice**([**1**,** 2**,** 3**,])**

layer\_depth **=** sorted**([**\

random**.**choice**([**32**,** 64**,** 128**,** 256**,** 512**,** 1024**])** \

**for** \_ **in** range**(**n\_conv\_layers**)])**

**for** i **in** range**(**1**,** len**(**layer\_depth**)):**

**if** layer\_depth**[**i**]** **<=** layer\_depth**[**i**-**1**]:**

layer\_depth**[**i**]** **=** 2 **\*** layer\_depth**[**i**-**1**]**

# Add the first Convolutional layer

model **=** tf**.**keras**.**Sequential**(**name**=**modelname**)**

**for** lyr **in** ModelFactory**.**get\_conv\_layer**(**layer\_depth**[**0**],** inshp**,** force\_stride\_size**=**fss**):**

model**.**add**(**lyr**)**

# Add zero or more additional convoluationa layer

**for** d **in** layer\_depth**[**1**:]:**

**for** lyr **in** ModelFactory**.**get\_conv\_layer**(**d**,** inshape**=None,** force\_stride\_size**=**fss**):**

model**.**add**(**lyr**)**

# Flatten

model**.**add**(**tf**.**keras**.**layers**.**Flatten**())**

# Add zero or more dense layers

n\_dense **=** random**.**choice**([**0**,** 1**,** 2**])**

**for** \_ **in** range**(**n\_dense **-** 1**):**

act\_fn **=** random**.**choice**([**'tanh'**,** 'relu'**,** 'sigmoid'**,** 'selu'**,** 'elu'**])**

layer **=** tf**.**keras**.**layers**.**Dense**(**\

random**.**choice**([**32**,** 64**,** 128**,** 256**,** 512**]),**

activation**=**act\_fn**)**

model**.**add**(**layer**)**

**if** random**.**choice**([True,** **False]):**

layer **=** tf**.**keras**.**layers**.**Dropout**(**0.25**)**

model**.**add**(**layer**)**

# Add the final Dense layer

model**.**add**(**tf**.**keras**.**layers**.**Dense**(**nlabels**,** activation**=**'softmax'**))**

**return** model

*@staticmethod*

**def** print\_model**(**model**):**

plot **=** tf**.**keras**.**utils**.**plot\_model**(**model**,** show\_shapes**=True,** expand\_nested**=True)**

IPython**.**display**.**display**(**plot**)**

**print(**model**.**summary**())**

*@staticmethod*

**def** generate\_model\_size**(**inshp**:**tuple**,** nlabels**:**int**,** modelname**:**str**,** max\_params**:**int**):**

**if** **(-**1 **==** max\_params**):**

**try:**

model **=** ModelFactory**.**generate\_model\_internal**(**\

inshp**,**\

nlabels**,**\

modelname**,**\

**False)**

**except:**

model **=** ModelFactory**.**generate\_model\_internal**(**\

inshp**,**\

nlabels**,**\

modelname**,**\

**True)**

ModelFactory**.**print\_model**(**model**)**

**return** model

**else:**

# Make 10 attempts to generate a model with the number of

# trainable params less than the one specified

**for** i **in** range**(**10**):**

gc**.**collect**()**

logging**.**debug**(**f"Attempt {i}"**)**

**try:**

model **=** ModelFactory**.**generate\_model\_internal**(**\

inshp**,**\

nlabels**,**\

modelname**,**

**False)**

**except:**

model **=** ModelFactory**.**generate\_model\_internal**(**\

inshp**,**\

nlabels**,**\

modelname**,**

**True)**

**if** model**.**count\_params**()** **<** max\_params**:**

ModelFactory**.**print\_model**(**model**)**

gc**.**collect**()**

**return** model

gc**.**collect**()**

**return** **None**

*@staticmethod*

**def** generate\_model**(**inshp**:**tuple**,** nlabels**:**int**,** modelname**:**str**,** max\_params**:**int**):**

"""

Tries to make a model with the number of trainable params less

than the specified number. If it is not able to do so, it will

create a model with a greater number regardless

"""

model **=** ModelFactory**.**generate\_model\_size**(**\

inshp**,**\

nlabels**,**\

modelname**,**\

max\_params**)**

**if** **None** **==** model**:**

model **=** ModelFactory**.**generate\_model\_size**(**\

inshp**,**\

nlabels**,**\

modelname**,**\

**-**1**)**

**return** model

## Methodology

First, a set of 10 base learners were randomly generated, and each of them were trained individually with the training set. Then they were used to predict the train and validation set, and their outputs stored, and combined by either of two methods:

Method 1:

The output prediction probability for each class from all the base estimators was summed, and the highest value was taken and the class was the predicted output

Method 2:

A second feature vector was constructed that had the following:

1. The output class probability for each class for each base learner (90 features, 9 classes x 10 base learners)
2. The summation of class probability for each class from all the base learners (9 features)

The total number of features here is 99. The above vector was created for both the training and the validation data set. Then a second machine learning model was trained on the above vector for the training data, and evaluated on the validation data.

## Need for Variability

For any ensemble, variability is very important and Brown et al. discuss this at length in their paper. [[9]](#footnote-9) Krogh and Vedelsby[[10]](#footnote-10) prove that the quadratic error of the ensemble estimator is guaranteed to be less than or equal to the average quadratic error of the base estimators. If the base estimators do not have any variability, they will predict the exact same thing, and the ensemble estimator will predict the same as the base estimators.

Nakano and Udea[[11]](#footnote-11) try to explain the need for variability in terms of bias-variation decomposition. Generally bias and variation in any network tend to work opposite to each other. They use three measures: the average bias of all the base learners, the average variance of all the base learners, and the averaged covariance of the base learners. The goal is to decrease the covariance without increasing the bias.

Ensemble learning generally takes one of two approaches:

1. Using a lot of low-variance high bias classifiers, and then using a statistical technique to combine them. Each high bias classifier outputs a prediction that is slightly better than random chance. Many of these learners can then be combined to form a very accurate predictor as long as there is variability in these learners. It can be mathematically shown that with enough of these learners, the prediction can come close to perfect accuracy. This technique is called boosting.
2. The other method is to use a lot of high variance low bias learners, and combine them in such a way to reduce their variance by using some method of combining their predictions. This is usually an averaging, but may involve a secondary learner with low variance. This is the approach we take in this assignment. Each CNN is a high variance learner and may tend to overfit. However, we can mitigate this overfitting by combining the output of multiple CNNs. Again, combining is only helpful if there is variation in the learners.

Sharkey and Sharkey[[12]](#footnote-12) describe the process as trying to approximate an actual distribution of the data P from a set of distributions P1, P2, …, Pn learnt by the individual base estimators, and pictorially they represent this as follows:

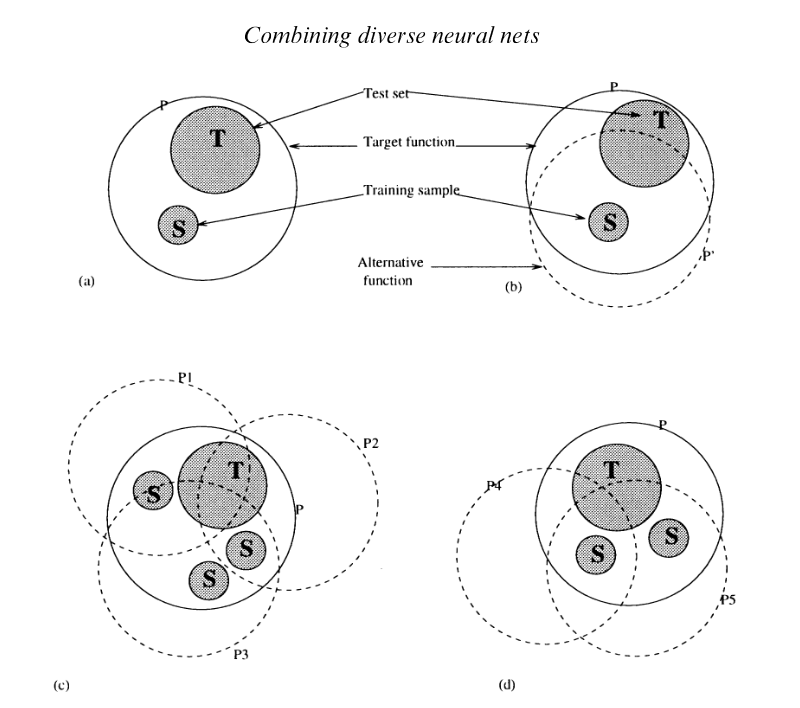


Figure 6 Sharkey and Sharkey, Combining Diverse Neural Nets

They argue that none of the individual estimators correctly estimate the distribution P of the test data P, but each of them overlaps a certain part of it, then combining multiple estimators covers the full space. This is seen in (c). If enough estimators are not used, and there is no variation in the estimators, we end up in a situation like (b) or (d) and the test data is not correctly classified.

Sharkey and Sharkey go further and describe four levels of diversity:

1. Level 1: For every input there is always a base learner that produces the correct output, and no inputs resulted in a failure by more than one base learner, then a simple majority vote suffices.
2. Level 2: For every input there are some failures, but the majority are always correct. Again, in this, a simple majority voting will produce the correct results.
3. Level 3: The actual function P is covered by the functions of each learner P1, P2, …, but most of the learners predict the incorrect output. In this case, a simple majority vote will no longer work. However, it is still possible to weight the outputs of the different learners in such a way that the correct output is produced. This is the approach we take in this assignment.
4. Level 4 – the functions of the individual base learners do not cover the actual function. In this case ensembles will not give the correct output.

As an aside, a similar philosophy also works in a wide variety of problems by employing variations in ideas from people rather than neural networks. [[13]](#footnote-13)

## Results

The first set of results presented are for the simple combination using sum of all probabilities and the use of a RandomForest algorithm to combine the predictions from all the learners (using the 99-features described above):

Done

val\_accuracies train\_accuracies model\_weights model\_names

0 0.806458 0.812708 7640457 Model - 0

1 0.759167 0.812292 127433 Model - 1

2 0.882500 0.907865 158345 Model - 2

3 0.125000 0.125000 6819977 Model - 3

4 0.653750 0.645885 28777 Model - 4

5 0.523750 0.549687 431561 Model - 5

6 0.644167 0.642500 15881 Model - 6

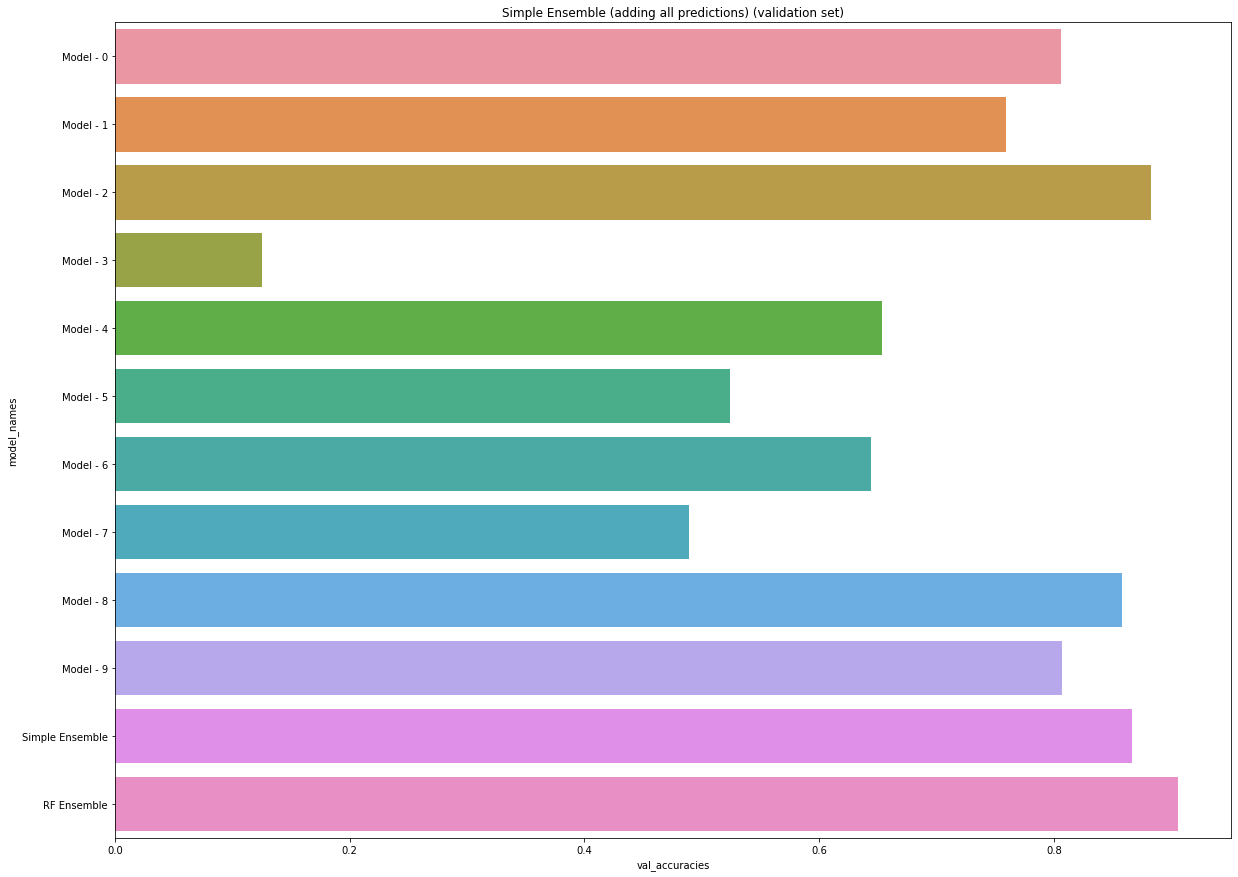
7 0.489375 0.530000 4196873 Model - 7

8 0.857917 0.910104 2362633 Model - 8

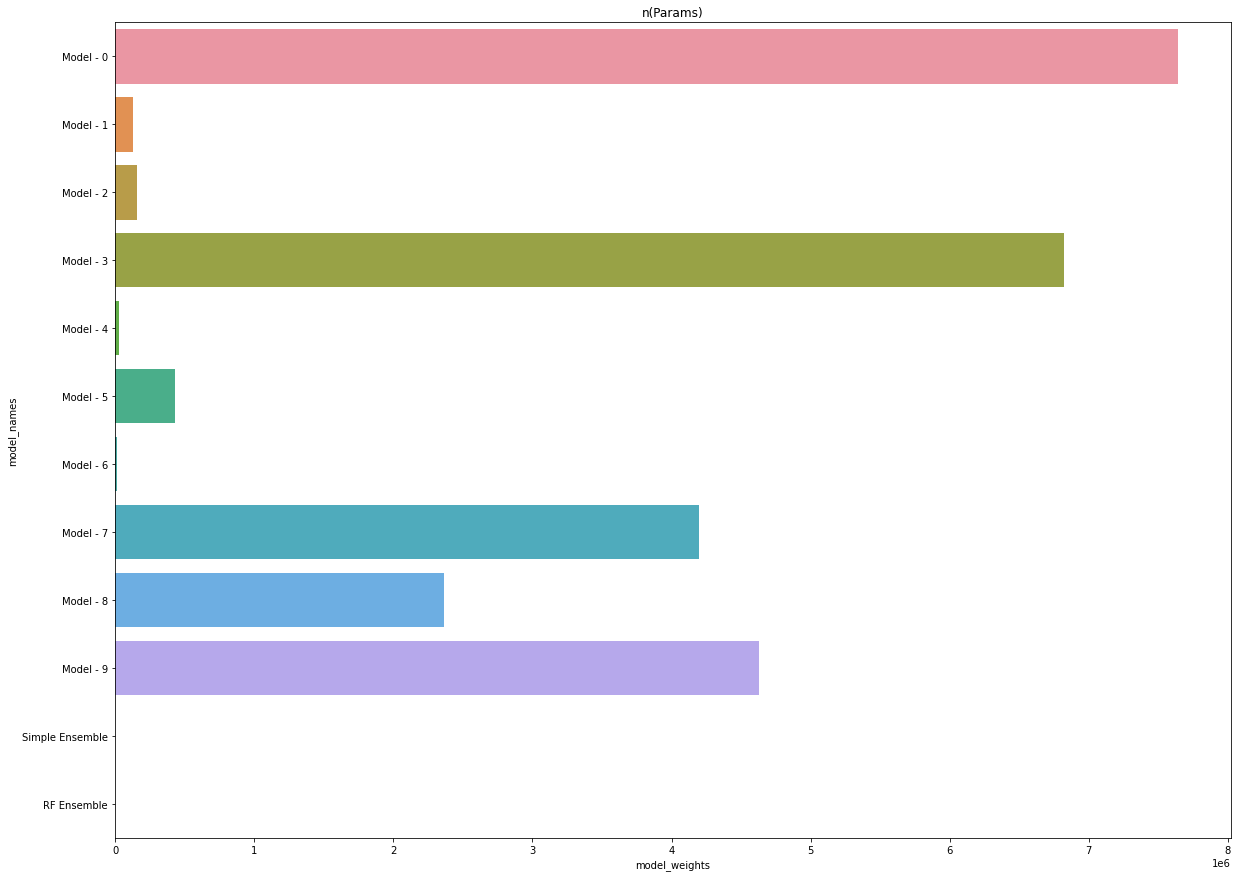
9 0.807083 0.907083 4631689 Model - 9

10 0.866875 0.911406 0 Simple Ensemble

11 0.905833 1.000000 0 RF Ensemble



We also present the number of parameters for each of the base models below to illustrate that it is not always the case that the model with the most number of parameters produces the best prediction. Model 0 has the maximum number of parameters, but doesn’t perform the best. As it may not be possible to know which neural network architecture may be best suited for the job in advance, using an ensemble of many neural networks may be a reasonable approach.



In these results we observe that the simple ensemble achieves an accuracy of 0.866 which is very good but it is not able to beat the best estimator Model 2 which achieves an accuracy of 0.8825. The random-forest based ensemble however beats the best estimator by achieving an accuracy of 0.90583*. Note that this accuracy is better even without data augmentation than all previously tested models, including much larger models where data was also augmented*.

When different classifiers in the second stage were compared, the results were that RandomForest performed the best. RandomForest is itself an ensemble. Surprisingly simple logistic regression also performed quite well when it was used to combine all the base learners, as did KNN. This would indicate that a relatively simple function can be used to combine many non-linear functions to produce a very accurate result.

DecisionTreeClassifier : 0.8602083333333334

KNeighborsClassifier : 0.89875

NearestCentroid : 0.87875

NaiveBayes : 0.8635416666666667

SVC : 0.9020833333333333

Random Forest : 0.9052083333333333

LDA : 0.8883333333333333

QDA : 0.874375

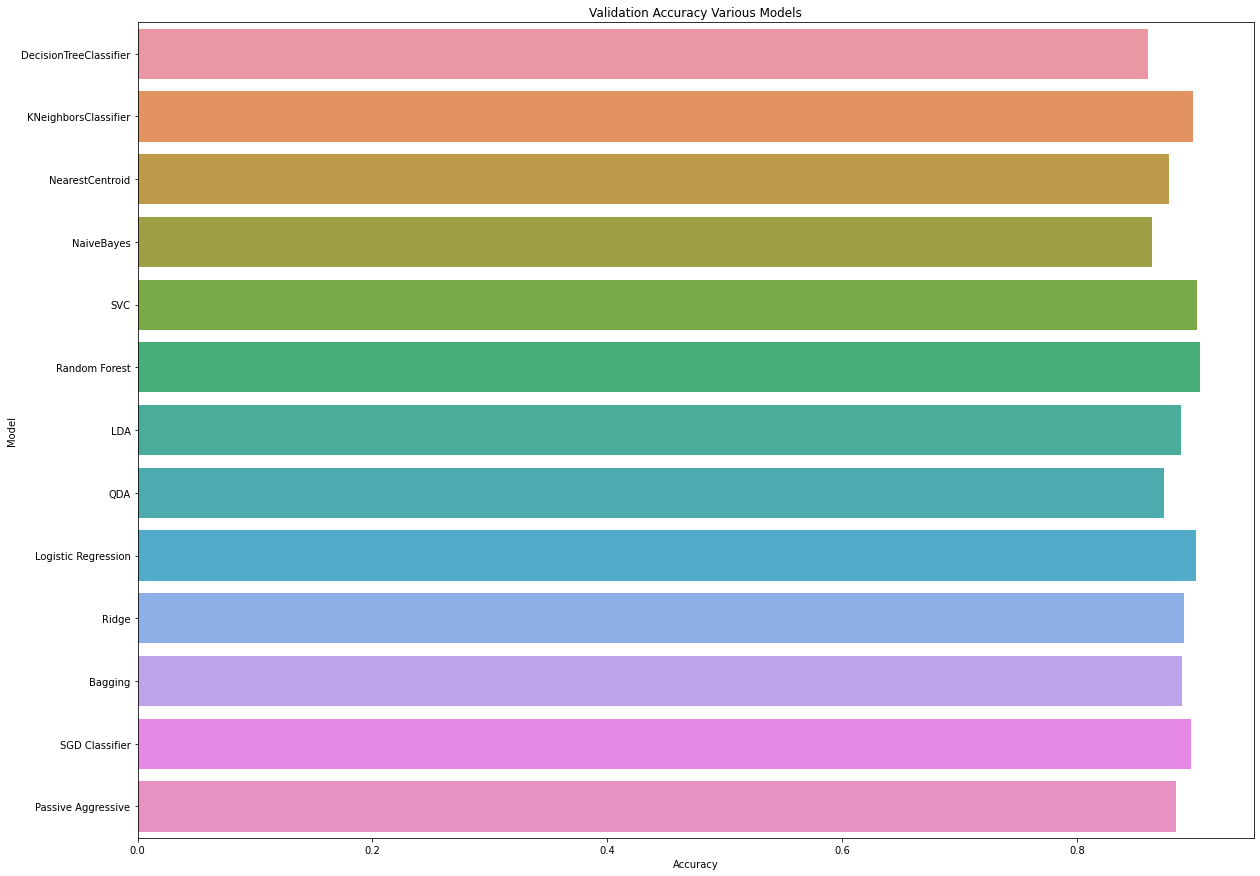
Logistic Regression : 0.9014583333333334

Ridge : 0.8914583333333334

Bagging : 0.8895833333333333

SGD Classifier : 0.896875

Passive Aggressive : 0.8841666666666667



## Future work

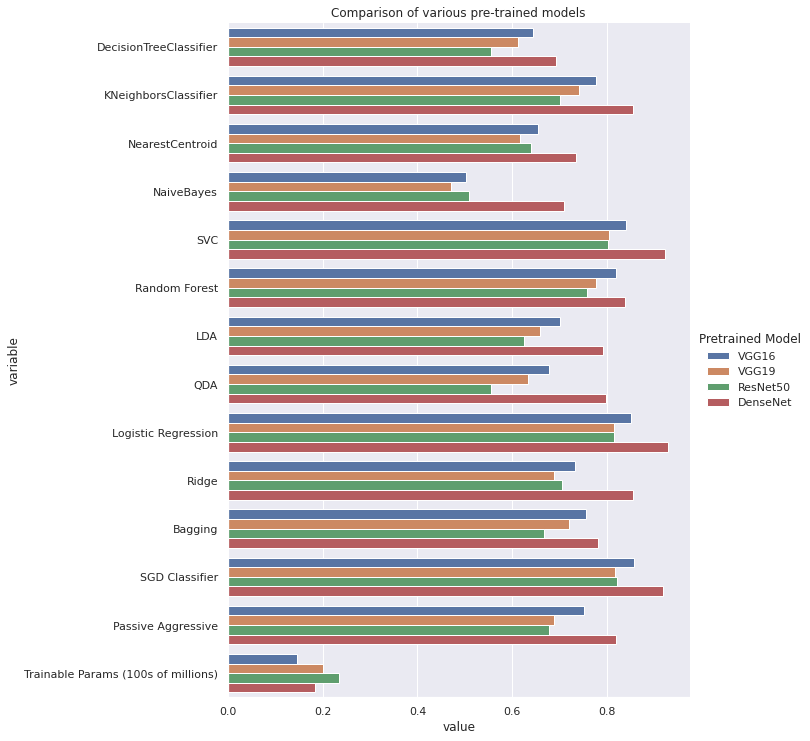
1. It would be worthwhile to check if adding a neural network (as opposed to RandomForest) in the second stage would produce better estimation
2. Each of the base classifiers are now giving their final predictions for each of the classes, and those are being combined. It would be worthwhile to experiment with combining an intermediate representation of the data rather than combining the final prediction would be better. This can be achieved by removing the last N layers of the base learners, and then combining the outputs. This is similar to what we do with transfer learning, except that in this case we can use multiple learners.

# Transfer Learning

## Pre-trained networks as feature extractors

The final layer from four pre-trained networks were dropped, and the network was treated as a feature extractor. The output from the model was passed through another machine learning algorithms.

Comparison of the various models gave this output:



The trends that we see are as follows:

1. Densenet performs the best from all the different networks
2. ResNet50 has the largest number of parameters, but its performance is the worst of all the four networks
3. VGG16 consitently performs better than VGG19, despite having fewer parameters.
4. Logistic Regression, SVC and SGD perform the best of all the models (upto an accuracy of 0.927). Random Forest, KNN and Passive Aggressive classifiers also have good results.

The key take-aways are this:

1. Densenet works as a very good feature extractor, followed by VGG-16. Without any training of the deep neural network, we were able to achieve an accuracy of 92.7%
2. A higher number of parameters in a DNN doesn’t necessarily mean a better accuracy.
3. The features produced by a pre-trained neural network can be effectively classified by relatively simple classifiers like Logistic Regression.

## Modify and Finetune

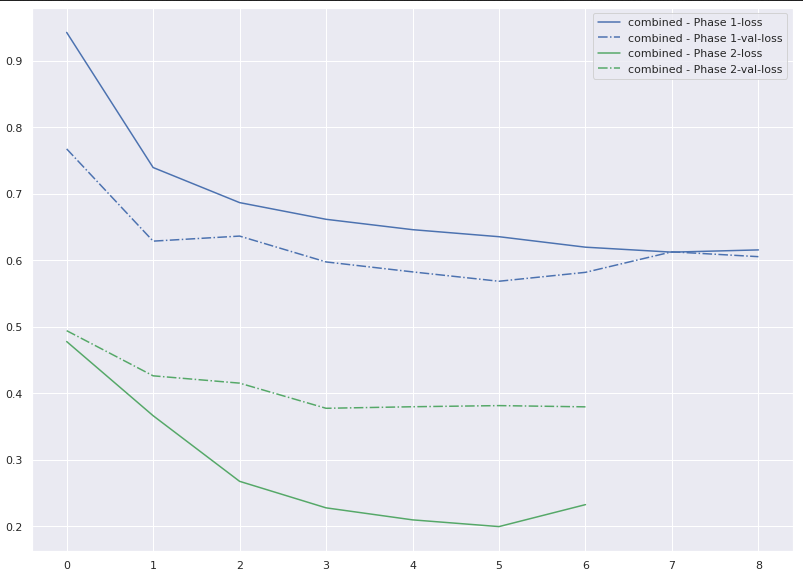
In this experiment, the last layer of the pre-trained network was dropped, and replaced with another neural network. The combined network was then trained by first freezing the pre-trained part of it. Once convergence was reached, the network was re-trained after unfreezing some of the layers of the pre-trained network, but with a very small learning rate. The results were then compared.

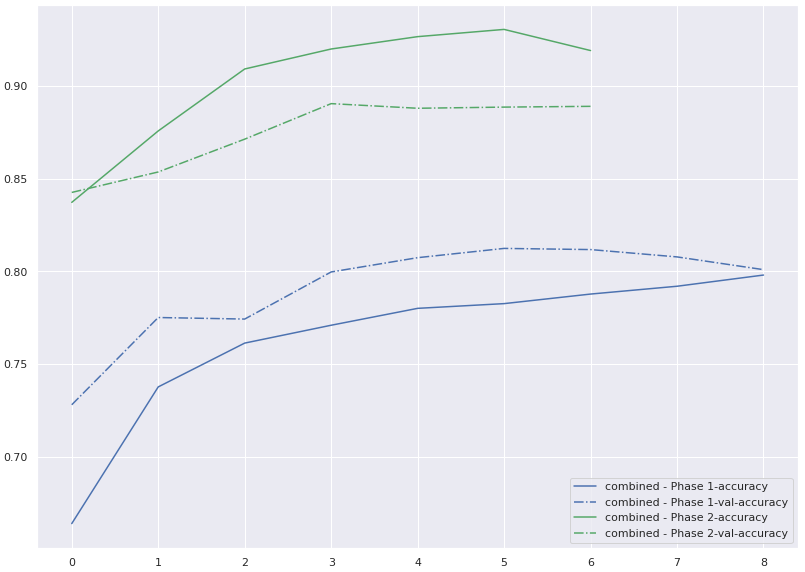
Three networks were used in combination with the pre-trained model. The table below describes the three different models used.

|  |  |
| --- | --- |
| Friendly Name | Architecture |
| Finetune0 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type) Output Shape Param #  =================================================================  flatten\_2 (Flatten) (None, 2048) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_2 (Dense) (None, 9) 18441  =================================================================  Total params: 18,441  Trainable params: 18,441  Non-trainable params: 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Finetune1 | Layer (type) Output Shape Param #  =================================================================  flatten (Flatten) (None, 2048) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dropout (Dropout) (None, 2048) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense (Dense) (None, 512) 1049088  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dropout\_1 (Dropout) (None, 512) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_1 (Dense) (None, 9) 4617  =================================================================  Total params: 1,053,705  Trainable params: 1,053,705  Non-trainable params: 0 |
| Finetune 2 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type) Output Shape Param #  =================================================================  flatten\_5 (Flatten) (None, 2048) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dropout\_11 (Dropout) (None, 2048) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_11 (Dense) (None, 1024) 2098176  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dropout\_12 (Dropout) (None, 1024) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_12 (Dense) (None, 512) 524800  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dropout\_13 (Dropout) (None, 512) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_13 (Dense) (None, 9) 4617  =================================================================  Total params: 2,627,593  Trainable params: 2,627,593  Non-trainable params: 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

In general, we see patterns like this between the first and the second phase, and there is a significant improvement in performance in the second phase of training.

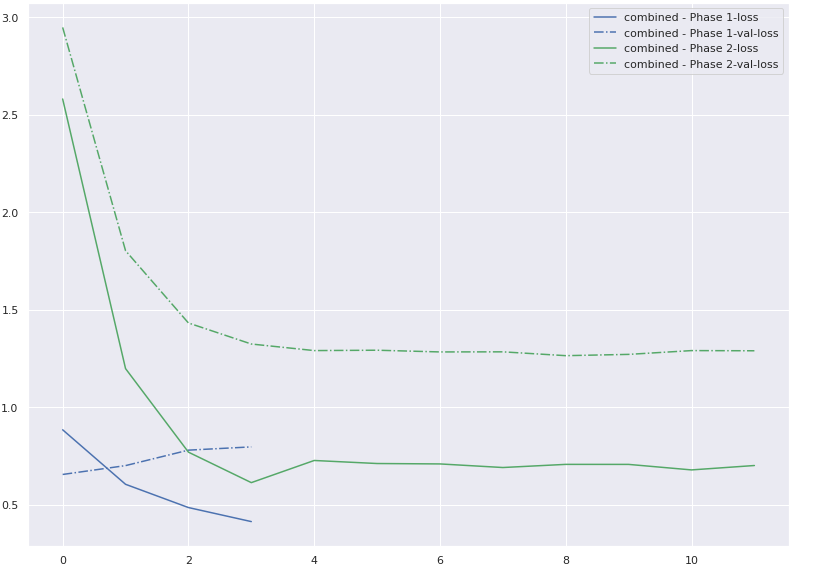
One tweak that was used throughout the second phase was to gradually decay the learning rate.



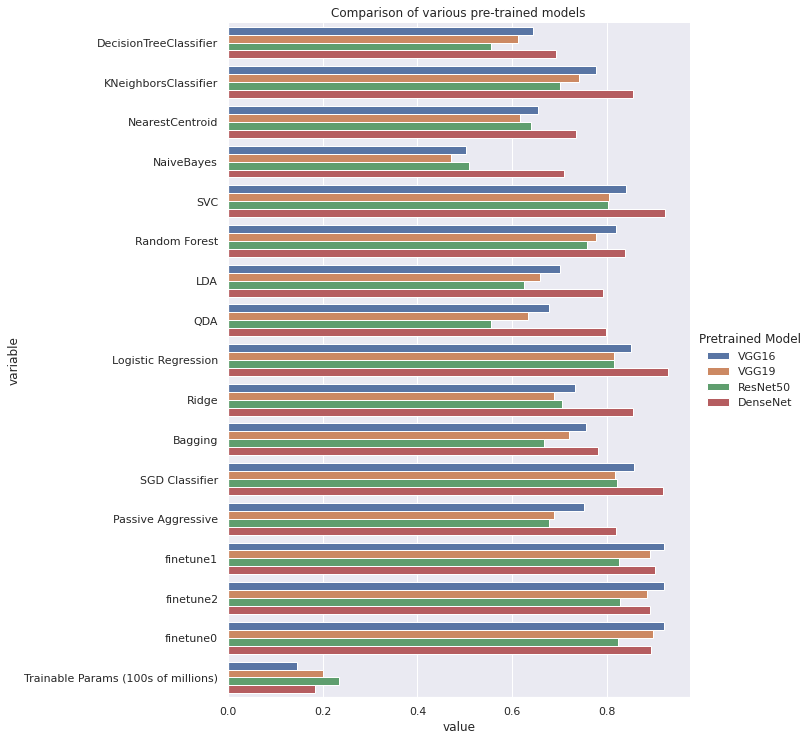


We see a difference between the final accuracy and loss of the first phase, and the starting accuracy and loss in the second phase because there has been one epoch of training before the first reading in phase 2 is taken.

However, it is the case sometimes that the pre-trained model performs worse after fine-tuning than the model we saved in phase 1. This may be because of getting stuck in a bad local minima. However, this is not a frequent occurrence.



When we compare all the three different custom networks that were appended at the end of the base learner, we get the following:



The observations we make are these:

1. Finetuning doesn’t produce better results than Linear Regression and SVC. However, this may be because of lesser number of epochs trained on. If more epochs are trained, or more layers in the pre-trained network fine-tuned, this may well beat linear Regresssion and SVC
2. We see that compared to traditional machine learning along with feature extraction, different networks behave differently. VGG-16 and VGG19 do not perform too well as feature extractors, but they perform very well with finetuning.
3. We see that after fine-tuning VGG-16 fares better than VGG-19.
4. We see that all the three networks that were added perform with similar accuracy numbers. This would suggest that the pre-trained network is already very good, and there is very little scope for the last few layers to learn.
5. Again, we see that its not necessarily the case that the model with the highest number of parameters is the most accurate.
6. Oveall, an accuracy of 0.9196 is achieved with VGG-16 and finetune 2.

# Capsule Networks

CNNs have excelled in the task of image recognition. One benefit of CNNs is that they are rotation and translation equivariant. Images in the CNN can be moved anywhere within the image, and they will still be categorized correctly. However, the translation invariance of CNN is also a problem. As an example, a human face is made out of two eyes, one nose and one mouth, and a CNN can learn to recognize those. However, if the eyes nose and mouth are moved to random locations in the image, the CNN will still recognize it as the same image. This problem comes out of the CNN operations because of two problems:

1. CNNs rely on convolutional layers learning base level features, which are combined into higher level features. Relationships between the base level features are not captured.
2. The use of max-pooling leads to loss of information which may be essential to the classification process.

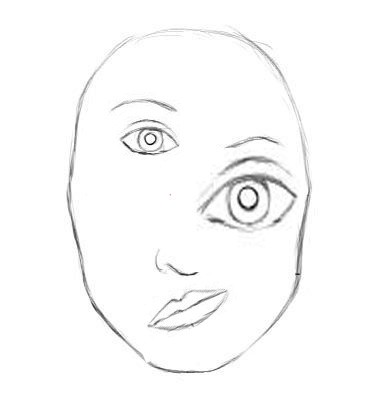


Figure 7 Capsule Networks: Source: The New Deep Learning Network, Aryan Misra

In the above image, the eyes, nose, mouth etc. form the lower level features, and they combine to give the higher level feature which is the face. A CNN will recognize this as a valid face, but a capsule network will be able to correctly infer that this is not a face.

Capsule Networks were proposed by Geoffrey Hinton and others as a principled way to solve this problem, and has undergone several iterations and addresses some of these problems. Some of the challenges where Capsule Networks outperform CNNs are:

* Finding repeating patterns in images in an unsupervised manner
* Separate occluded images, for example separate out occluded hand-written digits
* Viewpoint invariance and generalization, training data may have examples of images taken from one viewpoint perspective, but CapsNets can generalize and learn to recognize the same object photographed from another viewpoint
* If the parts of the image are separated and reorganized, they should not be recognized as the whole

Some other benefits of capsule networks are as follows:

* They typically have a smaller number of learnable parameters
* They can potentially learn from a fewer number of training examples
* They usually outperform CNNs in image recognition tasks
* They are more resilient against adversarial attacks
* They have been used in novel ways to detect adversarial attacks[[14]](#footnote-14)

The key in all these tasks lies in the observation that the position and structure of the lower level features and the relationships between them.

While CNNs produce a scalar output, CapsNets produce a vector output. In some sense they are inspired by the older graphics processing community which preferred to convert a pixel-space problem to a vector-space problem before working on it.

In Capsule Networks, a coordinate frame is added for each part that makes up the whole. This coordinate frame encodes the following:

1. Rotation
2. Scale
3. Position

Coordinate frames of parts must be consistent with each other to infer the whole, and the goal is to learn relationships of the coordinate frames with each other.

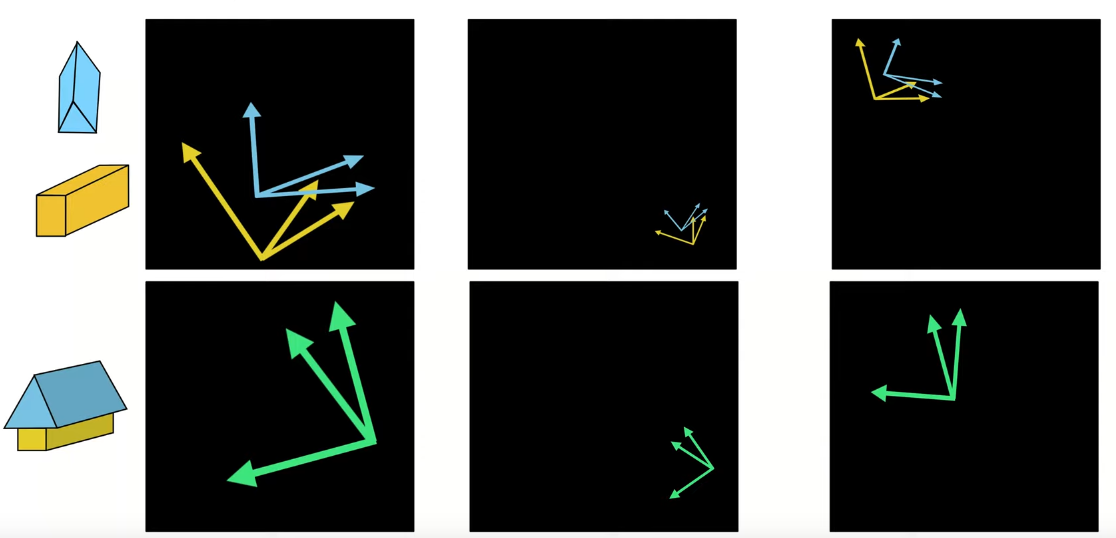


Figure 8 Source: Introduction to Capsule Networks, Sara Sabour, presented to UCF at CRCV

In the above picture, a house is made up of roof and four walls. However, the roof and the four walls have to be in the correct alignment to make up the house. We can say that every object in the picture has a direction, magnitude and position which are represented by the vectors. The roof and walls have to agree with each other to be classified as a house.

There are two concepts that are of interest here:

**Coordinate Frame:** It is a vector or matrix   
**Frame Relationship:** It is a transformation matrix that transforms one frame to another, typically lower level features are transformed to see their suitability for higher level features  
**Capsule:** A capsule is a group of neurons that represent whether an entity exists, and its coordinate frame if it does. A capsule can be a single dimensional vector or a multi-dimensional matrix. Capsules are a group of neurons that learn a semantically meaningful concept.

The general mechanism for a capsule network to work is to fine agreement between capsules. Lower level capsules are linked to higher level capsules. Each higher level capsule recognizes a different higher level concept. For example, in the following diagram, the lower level capsules recognize the different lines, and the higher level capsules learn the digits.

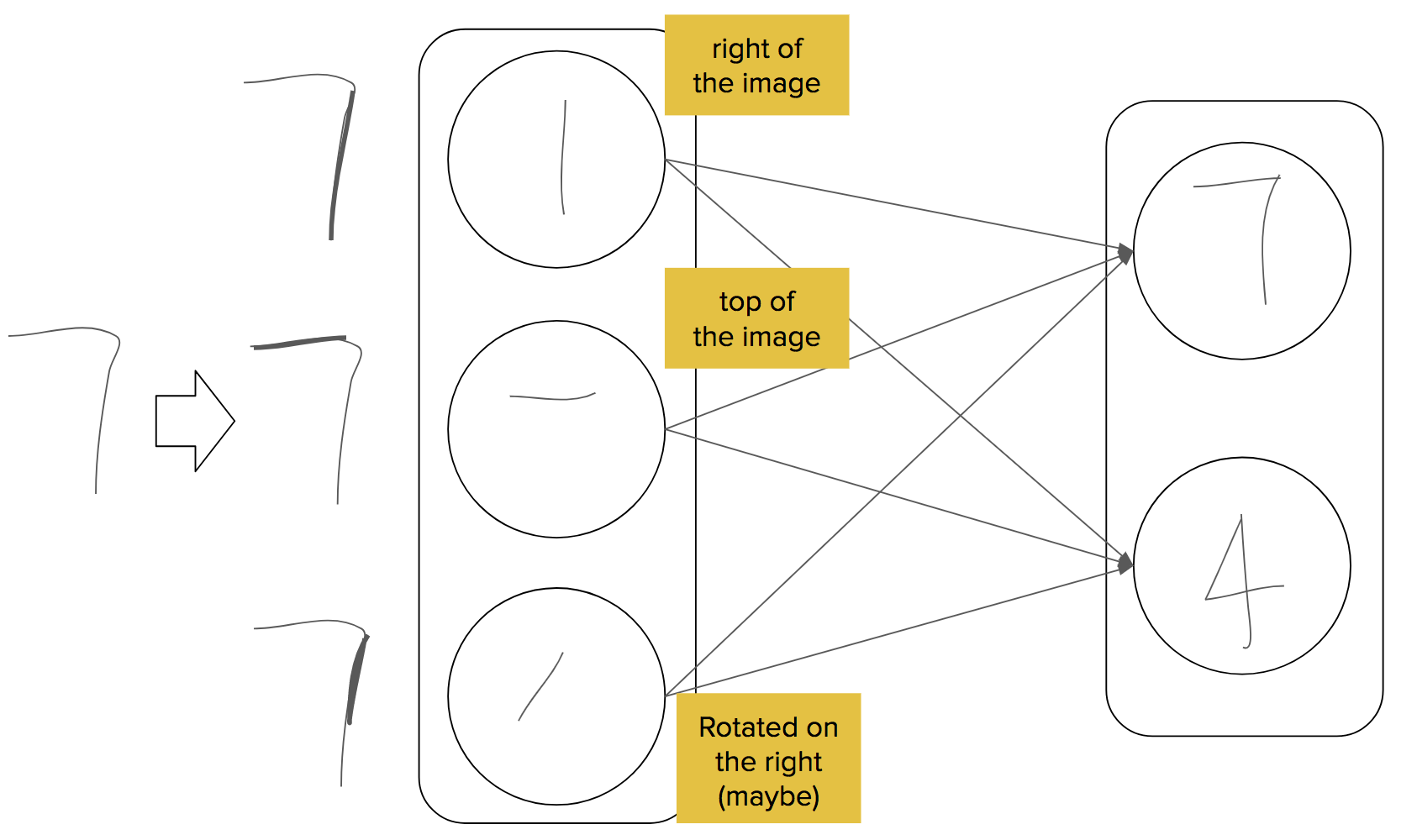


Figure 9 Source: The New Deep Learning Network, Aryan Misra

Each of the lower level capsules transform their coordinate frames and send their ‘predictions’ to the higher level features. The higher-level feature is predicted only if the predictions match. This process is called agreement. The goal is to create a parse tree where each lower level capsule is assigned to one parent, and after the process of agreement, each lower level capsule sends it output only to that higher-level capsule that is assigned as its parent. Any non-leaf capsule that doesn’t have children are pruned. After the process of agreement a ‘parse-tree’ is created, and the goal of learning is to find a good parse tree.

Assignment of parents is a layer-wise non-linear operation and parents compete with each other for children. Transformations that take the output of each lower level capsule and map it to a higher-level capsule are the only learnable parameters. Agreement is said to be reached when lower level capsules have to perform similar transformations to map with the higher level capsule.

In the paper “Dynamic Routing between capsules” [[15]](#footnote-15), the following measures are specified

1. Agreement Measure: In dynamic routing, this is the cosine similarity. Cosine similarity is used because if vectors are collinear, then the cosine distance is highest, and if vectors are orthogonal, cosine similarity is lowest.
2. Existence Probability: In dynamic routing, this is the norm of the coordinate frame

A RANSAC style algorithm is used to find the routing (which is another name for assigning children capsules to their parents). The following steps are iteratively performed:

1. Find the dot product of each capsule with the next level capsule
2. For capsules which can be unambiguously assigned, assig them to the higher level capsule with the highest dot product. This basically weeding out outliers. Remove these capsules from all other parents
3. Repeat the process with a smaller number of child capsules

Typically, three iterations are enough to produce a good parse tree.[[16]](#footnote-16)

In dynamic routing, the norm of the coordinate frame is used as a measure of existence. However, if a coordinate frame is scaled, it will have a larger norm. Since existence must be a probability, a squash function is applied to map it to a range between 0 and 1. This is a problem since information is lost in the process.

The next improvement to this process is EM routing[[17]](#footnote-17) which tries to address the above problem. In EM routing, the norm is made a part of the coordinate frame, and existence is a separate parameter. Since the norm is now a part of the coordinate frame, Euclidean distance is used. The algorithm used to find agreement is very similar to the Expectation Maximization step in Gaussian Mixture Models. The idea is to find a set of Gaussians that best describe clusters of lower level features, and this step is similar to a KNN cluster. Clusters that are tight (where the standard deviation of the fitted Gaussian is low) are assigned to that parent. Clusters that are loose are not assigned to that parent.

Essentially, the above two algorithms are iterative, and therefore computationally intensive. Hahn, et al. have attempted to replace this iterative algorithm with MLPL based routing, which makes this entire process free from iterations, but this comes at the expense of new learnable parameters.[[18]](#footnote-18)

Several other routing algorithms have also been proposed

1. Attention based routing[[19]](#footnote-19)
2. Multi Modal capsules[[20]](#footnote-20)
3. Generic Routing[[21]](#footnote-21)

While capsule networks are much better than CNNs at image classification tasks, as well as image reconstruction tasks, they suffer from certain drawbacks

1. They are computationally intensive
2. As the number of classes grow, the number of parameters also grow, hence they may not fit into memory
3. Due to processing constraints it is hard to apply them to multi-dimensional data like video, although there has been some work in scaling up capsule networks for video by applying pooling between capsules[[22]](#footnote-22)

1. Inoue, Hiroshi. "Data augmentation by pairing samples for images classification." *arXiv preprint arXiv:1801.02929* (2018). [↑](#footnote-ref-1)
2. Zhong, Zhun, et al. "Random erasing data augmentation." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. No. 07. 2020. [↑](#footnote-ref-2)
3. Shorten, Connor, and Taghi M. Khoshgoftaar. "A survey on image data augmentation for deep learning." *Journal of Big Data* 6.1 (2019): 1-48. [↑](#footnote-ref-3)
4. Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014. [↑](#footnote-ref-4)
5. Lecture on Zero-Shot Learning - Dr. Timothy Hospedales, https://www.youtube.com/watch?v=jBnCcr-3bXc [↑](#footnote-ref-5)
6. McCartney, Ben, et al. "A zero-shot learning approach to the development of brain-computer interfaces for image retrieval." *Plos one* 14.9 (2019): e0214342. [↑](#footnote-ref-6)
7. Xie, Huang, and Tuomas Virtanen. "Zero-shot audio classification based on class label embeddings." 2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA). IEEE, 2019. [↑](#footnote-ref-7)
8. Wang, Yaqing, et al. "Generalizing from a few examples: A survey on few-shot learning." ACM Computing Surveys (CSUR) 53.3 (2020): 1-34. [↑](#footnote-ref-8)
9. Brown, Gavin, et al. "Diversity creation methods: a survey and categorisation." *Information fusion* 6.1 (2005): 5-20. [↑](#footnote-ref-9)
10. A. Krogh, J. Vedelsby, Neural network ensembles, cross validation, and active learning, NIPS, 7 (1995), pp. 231-238 [↑](#footnote-ref-10)
11. N. Ueda, R. Nakano, Generalization error of ensemble estimators, in: Proceedings of International Conference on Neural Networks, 1996, pp. 90–95 [↑](#footnote-ref-11)
12. A. Sharkey, N. Sharkey, Combining diverse neural networks, The Knowledge Engineering Review, 12 (3) (1997), pp. 231-247 [↑](#footnote-ref-12)
13. Surowiecki, James. The wisdom of crowds. Anchor, 2005. [↑](#footnote-ref-13)
14. Frosst, Nicholas, Sara Sabour, and Geoffrey Hinton. "Darccc: Detecting adversaries by reconstruction from class conditional capsules." *arXiv preprint arXiv:1811.06969* (2018). [↑](#footnote-ref-14)
15. Sabour, Sara, Nicholas Frosst, and Geoffrey E. Hinton. "Dynamic routing between capsules." *arXiv preprint arXiv:1710.09829* (2017). [↑](#footnote-ref-15)
16. <https://www.youtube.com/watch?v=zRg3IuxaJ6I&t=368s>, Introduction to Capsules by Sara Sabour [↑](#footnote-ref-16)
17. Hinton, Geoffrey E., Sara Sabour, and Nicholas Frosst. "Matrix capsules with EM routing." *International conference on learning representations*. 2018. [↑](#footnote-ref-17)
18. Hahn, Taeyoung, Myeongjang Pyeon, and Gunhee Kim. "Self-routing capsule networks." (2019). [↑](#footnote-ref-18)
19. Choi, Jaewoong, et al. "Attention routing between capsules." *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 2019. [↑](#footnote-ref-19)
20. McIntosh, Bruce, et al. "Multi-modal capsule routing for actor and action video segmentation conditioned on natural language queries." *arXiv preprint arXiv:1812.00303* (2018). [↑](#footnote-ref-20)
21. Lenssen, Jan Eric, Matthias Fey, and Pascal Libuschewski. "Group equivariant capsule networks." *arXiv preprint arXiv:1806.05086* (2018). [↑](#footnote-ref-21)
22. Duarte, Kevin, Yogesh S. Rawat, and Mubarak Shah. "Videocapsulenet: A simplified network for action detection." *arXiv preprint arXiv:1805.08162* (2018). [↑](#footnote-ref-22)