Assignment 2

Group No 20

Tanay Dixit (EE19B123) , Atharva Chougule (CS19B016) , Kaustubh Miglani (CS19B060)

30th March , 2022

CS6910

Some common things that we did for all models

We used Adam optimizer for the models as it converges faster. We save the checkpoint based on the best validation accuracy.

1 Task 1

1.1 Data Compression with Autoencoder and PCA

To decided to what output dimension should we compress the given input to we found the eigen values of the dataset and plotted the graph of % cumulative variance of data vs number of principal components.

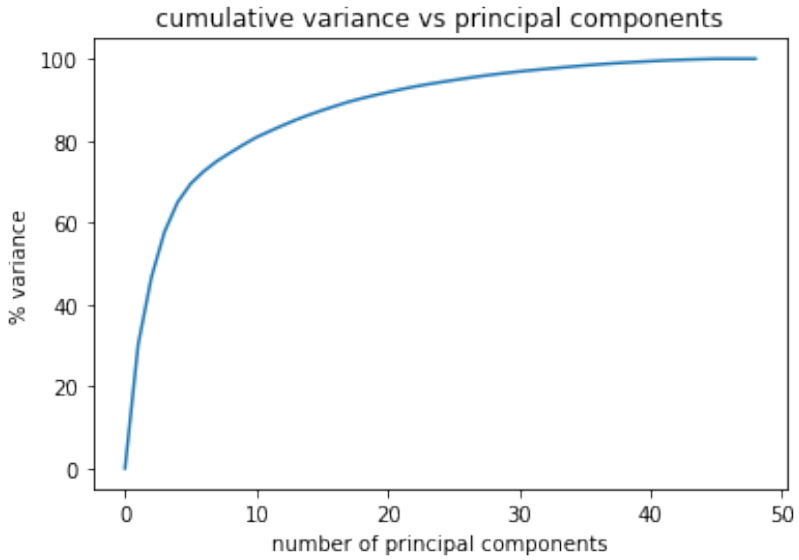


Figure 1: Cumulative Variance vs Principal Components

From the graph we chose the value for output dimension as 10, as the cumulative variance reaches the value of 80% when the number of principal components become 10, and also this value comes after the 'elbow' in the above graph. As 10 comes after the elbow the % cumulative variance won't increase much with increase in number of principal components as the slope is low. Hence we decided to take 10 as our length for output dimension.

1.1.1 Autoencoder

We used an autoencoder with 2 nonlinear layers for compressing the 48-D input data to 10-D vectors. After experimenting we found that using L1Loss i.e Mean average error as the objective function gave better results than using MSE Loss. The reason why L1Loss worked better was because the range of values in the input data is very large, and MAE usually performs better than MSE on data with large range of values. We also found that using RELU activation function for nonlinear layers in autoencoder gave much more better results that Tanh or Sigmoid activation functions. After fixing above things we tuned the autoencoder with hyperparameters as number of nodes in hidden layers of auto encoder.

We choose the best parameters from figure 12 and train our final model. The training

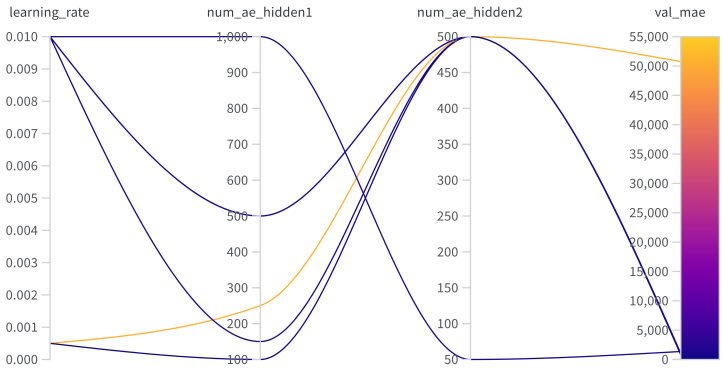
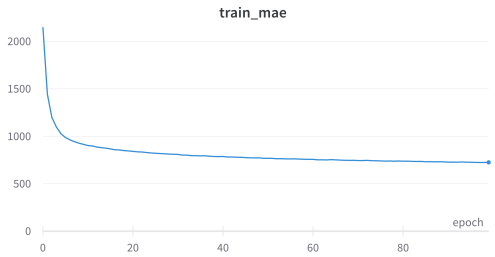


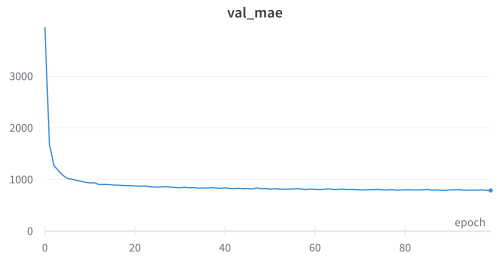
Figure 2: Parameter Tuning

curves are in 3a and 3b

Observations: We find that the autoencoder compresses the input to 10-D with a train



(a) Train Mean Absolute Error



(b) Validation Mean Absolute Error

error(MAE) of 725.534 and val error(MAE) of 788.181.

1.1.2 PCA

We used pca to compress the given 48-D data to 10-D. We used 80-20 split as mentioned in the assignment, and then we calculated the reconstruction errors for the pca model.
Observations: We got the following values of reconstruction errors on pca :- Train error(MAE) of 1198.397 and Val error(MAE) of 1215.896

Comparision of PCA and Autoencoder:

1. We found that both PCA and Autoencoder give a good representation of input data of large dimension by reducing it to lower dimension space.
2. The val errors(MAE) for reconstructed data for PCA and Autoencoder are 1215.896 and 788.181 respectively. Given that the range of values in input are from 0 to 35000, these errors are low compared to the input values. Hence we can conclude that both PCA and Autoencoder are good data compression models.
3. From the errors it is evident that Autoencoder is a much superior model as compared to PCA for data compression as the error of 788.181 in case of Autoencoders is almost 27% less than the error of 1215.896 in case of PCA.
4. Autoencoders perform much better than PCA because Autoencoders are able to learn non-linear features in input data as they contain non-linear hidden layers, but PCA can only represent linear relationships in input data as it is a linear model.

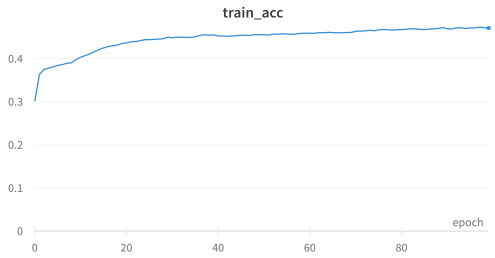
1.2 MLFFN on compressed data

We used the Autoencoder and PCA model that we trained above for compressing the given data to 10-D vecctors and we then pass this 10-D vectors to a multi layer feed forward neural network(MLFFN) for classification. Our network consisted of 2 hidden layers and a tanh activation. The best hyperparameters for this network were 50 and 20 nodes respectively.

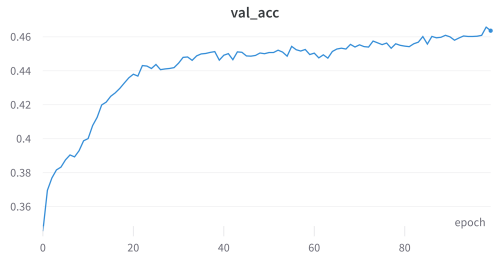
We got the following results :-

1.2.1 MLFFN on Autoencoder compressed data

:



(a) Train accuracy



(b) Val accuracy

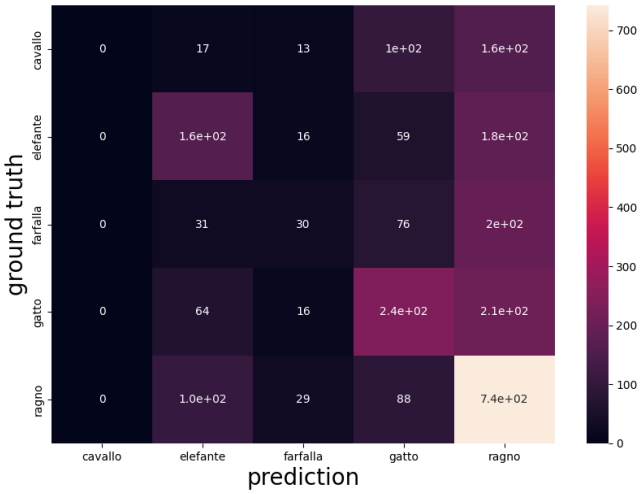
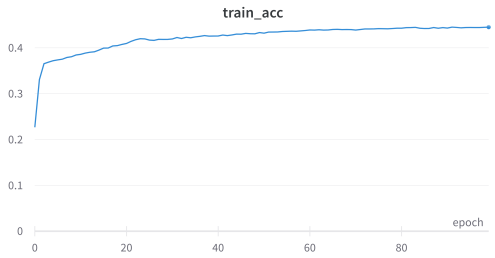


Figure 5: Confusion Matrix for MLFFN on Autoencoder compressed data

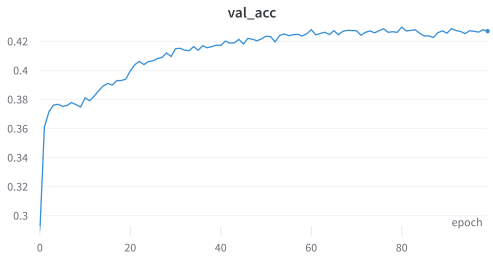
We got a training accuracy of 47.1% and a validation accuracy of 46.3% by passing the Autoencoder compressed data thorugh a MLFFN.

1.2.2 MLFFN on PCA compressed data

:



(a) Train accuracy



(b) Val accuracy

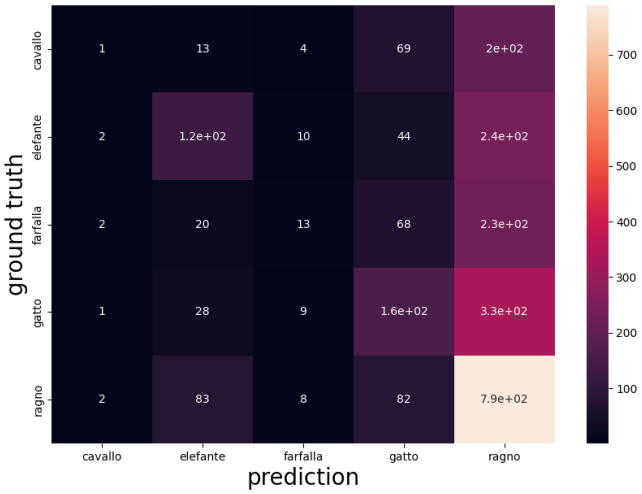


Figure 7: Confusion Matrix for MLFFN on PCA compressed data

We got a training accuracy of 44.5% and a validation accuracy of 42.7% by passing the PCA compressed data thorugh a MLFFN.

Observations:

1. We find that using MLFFN for classification on Autoencoder compressed data gives better results than using MLFFN for classification on PCA compressed data. The first gave an accuracy of 46.3% while the latter gave an accuracy of 42.7%.

2. From the confusion matrix we observe that both the models are almost extremely accurate in predicting images of classes 'elefante', 'gatto' and 'ragno', but fail completely in predicting images of classes 'carvallo' and farfella'.

3. One of the main reason for failure of above models in classyfying the two classes can be that we are using histogram vectors as our inital input to the models, but histogram vector of the whole image is just a representaion of color intensity of pixels in the image as a whole.

4. Thus two images which gave more or less same color scheme we have very close histogram vectors, and hence the two models will fail in calssifying the images correctly. We have no information about local features in these histogram vectors.

5. Thus it is possible that both the missclassifed classes 'carvallo' and farfella' have the same color scheme as images in class 'ragno' and hence all three classes have almost same representations of histogram vectors and as the number of datapoints in 'ragno' is much much larger than 'carvallo' or 'farfella' all the images in these 3 classes are classified as 'ragno' which is what we see in the confusion matrix.

2 Task 2

2.1 Data Compression with 3 Autoencoders and fine tuning Stacked Autoencoder

2.1.1 Training the Autoencoders

We decided that we will compress the given 48-D histogram vectors to 10-D vectors using stacked encoder with 3 autoencoder layers, as we experimented with other lower dimensions like 5/7/10 and noticed that the best results are obtained by using 10 dimensions. Since we have to compress 48D to 10D after using 3 encoding parts, so we decided that at each stage we will reduce the dimension by approximately $(48-10)/3 = approx\ 13$. Therefore the first autoencoder reduces 48D input to 35D, the second one does from 35D to 22D and the third one from 22D to 10D.

2.1.2 Fine Tuning the autoencoder and comparison

We use the values of weights obtained in the above mentioned training of autoencoders and then fine tune it . We compare the stacked autoencoder followed by fine tuning with another model which has random initialized values of weights and observe that this performs much better than using random initialized weights . We also conclude that this gives a better 10 D representation of the data as compared to just using 1 autoencoder layer or even PCA (As seen in Task 1) because it has more layers of non linearity involved. So it can actually learn a better 10D representation of the input .

2.2 Final values of the hyperparameters for each of the autoencoders

Autoencoder number	Input	Hidden Layer 1	Hidden Layer 2	Output
1	48	500	200	35
2	35	250	200	22
3	22	200	100	10

2.2.1 Things followed for Autoencoder 1

We used an autoencoder with 2 nonlinear layers. After experimenting we found that using L1Loss i.e Mean average error as the objective function gave better results than using MSE Loss. The reason why L1Loss worked better was because the range of values in the input data is very large, and MAE usually performs better than MSE on data with large range of values. We also found that using RELU activation function for nonlinear layers in autoencoder gave much more better results that Tanh or Sigmoid activation functions. We reduced the 48D input to 35D in this step. After fixing above things we tuned the autoencoder with hyperparameters as number of nodes in hidden layers of auto encoder.

We choose the best parameters from figure 12 and train our final model. The training

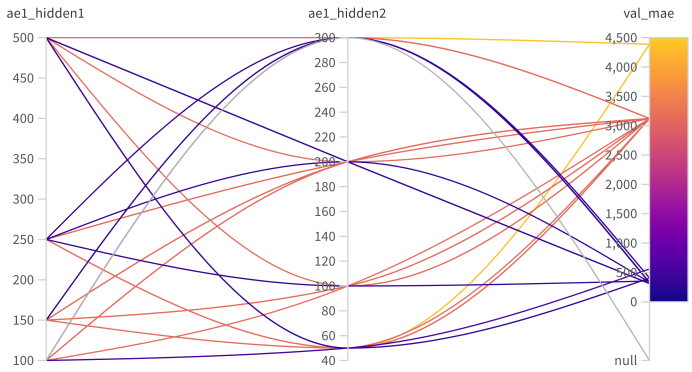
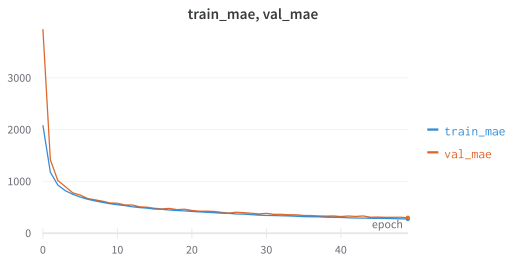
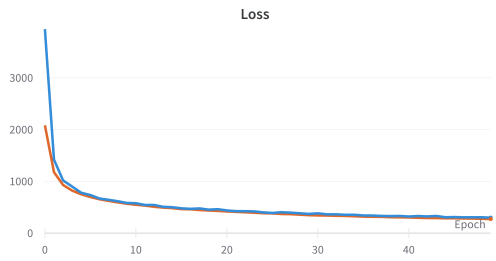


Figure 8: Parameter Tuning

curves are in 9a and 9b



(a) Accuracy Plots



(b) Loss Plots

2.2.2 Things followed for Autoencoder 2

We used an autoencoder with 2 nonlinear layers. After experimenting we found that using L1Loss i.e Mean average error as the objective function gave better results than using MSE Loss. We also found that using RELU activation function for nonlinear layers in autoencoder gave much more better results that Tanh or Sigmoid activation functions. We reduced the 35D input to 22D in this step. After fixing above things we tuned the autoencoder with hyperparameters as number of nodes in hidden layers of auto encoder.

We choose the best parameters from figure 12 and train our final model. The training

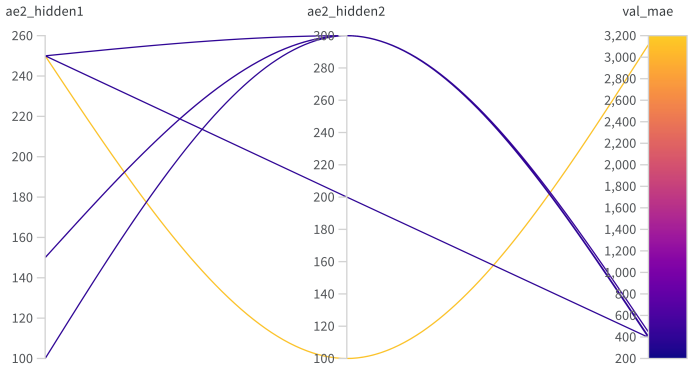
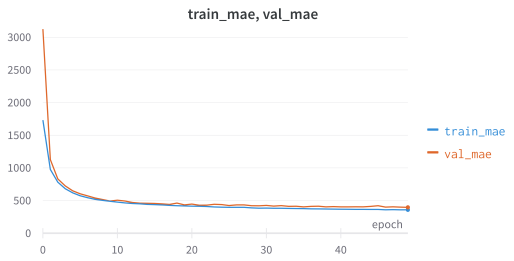
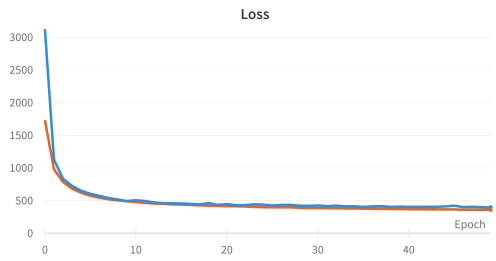


Figure 10: Parameter Tuning

curves are in 11a and 11b



(a) Accuracy Plots



(b) Loss Plots

2.2.3 Things followed for Autoencoder 3

We used an autoencoder with 2 nonlinear layers. After experimenting we found that using L1Loss i.e Mean average error as the objective function gave better results than using MSE Loss. We also found that using RELU activation function for nonlinear layers in autoencoder gave much more better results that Tanh or Sigmoid activation functions. We reduced the 22D input to 10D in this step. After fixing above things we tuned the autoencoder with hyperparameters as number of nodes in hidden layers of auto encoder.

We choose the best parameters from figure 12 and train our final model. The training

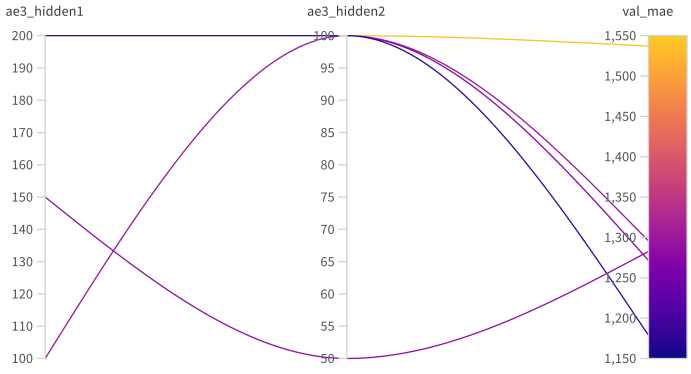
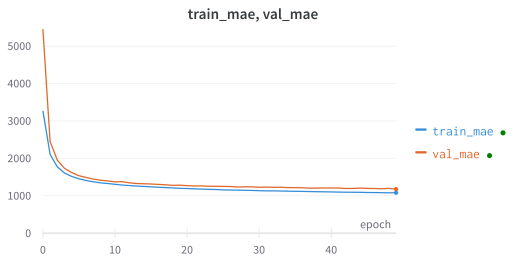
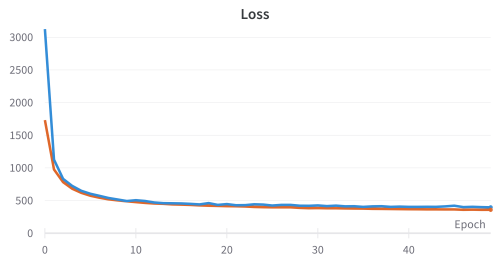


Figure 12: Parameter Tuning

curves are in 13a and 13b



(a) Accuracy Plots

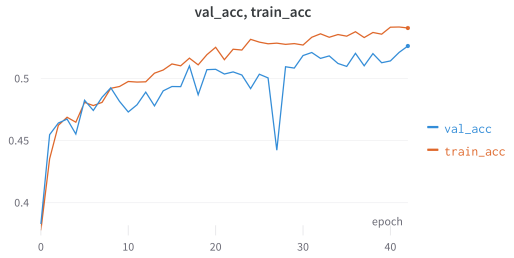


(b) Loss Plots

2.3 Fine tuning the parameters

We fine tuned the model obtained with the values of weights as that obtained in previous steps and we get the following train and validation accuracy plots and the following result for confusion matrix.

2.3.1 Accuracy and Loss plots



(a) Accuracy Plots

(b) Loss Plots

2.3.2 Confusion Matrix

The training accuracy was 0.547 and validation accuracy was found to be 0.534.

2.4 The model with randomly initialized weights

We trained the model with same hidden layers and number of nodes in each layer with randomly initialised weights. We obtained an accuracy of 38.9% with randomly initialized

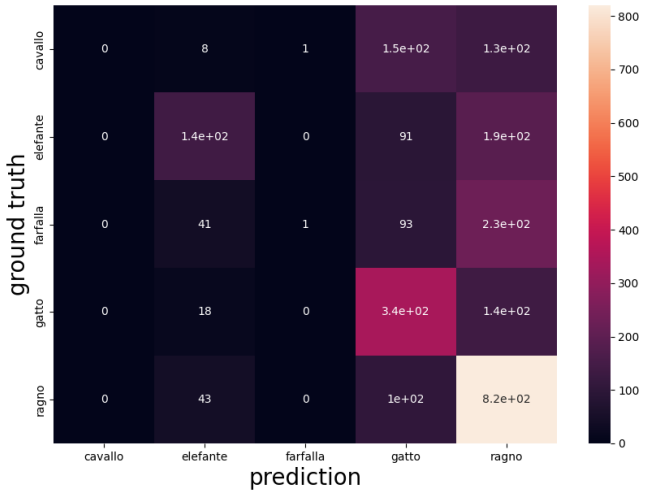


Figure 15: Confusion Matrix for stacked Autoencoder

weights. We obtain worse accuracies than using the weights which were trained using the stacked autoencoder. This helps provide empirical evidence that pre-training the autoencoder helps.

2.5 Observations

1. We find that stacked autoencoder works better than normal autoencoder data compression followed by MLFFN.

2. Stacked autoencoder gave an accuracy of 53.4% for classification while normal autoencoder + MLFFN model had given an accuracy of 46.3%.

3. We also trained the model wothout pre-training the three autoencoders and we directly randomly initalialized the weights which gave us an accuracy of 38.9%, which was very less than the model where we use pre-trained weights which gave 53.4%.

4. Thus pre-training phase of stacked autoencoder is an important step for the model.

5. But similar to task1 we find from confusion matrix that the model fails completely in distinguishing the images from 3 classes 'carvallo' , 'farfella' and 'ragno'.

6. Here too the reason of this failure can be attributed to the fact that images in these classes may have similar color schemes and hence will have similar representation of histogram features and hence all the 3 classes are classified as 'ragno' as'ragno' contains more number of data points in its dataset compared to two other classes.

7. This observation that histogram features fail to represent images with similar color schemes calls for the need that a stronger model for image classification is needed which can learn the local features in the image and not only use global features like the models and task1 and task2 did.

3 Task 3

3.1 VGGNet

We make use the `torchvsion` library to load the pretrained $vgg16$ model. Specifically we freeze the CNN parameters of the VGG model and extract the deep CNN features and feed it into another trainable feed forward dense network with $ReLU$ activations.

3.1.1 Training

The hyperparameters we could tune were only the batch size, learning rate and number of hidden nodes in the dense layers. We observe that changing batch size had minimal effects on performance hence we decided to use a standard value of 64. We experiment with 4 learning rate values, while varying the other hyperparameters as shown in the figure and observe the following.

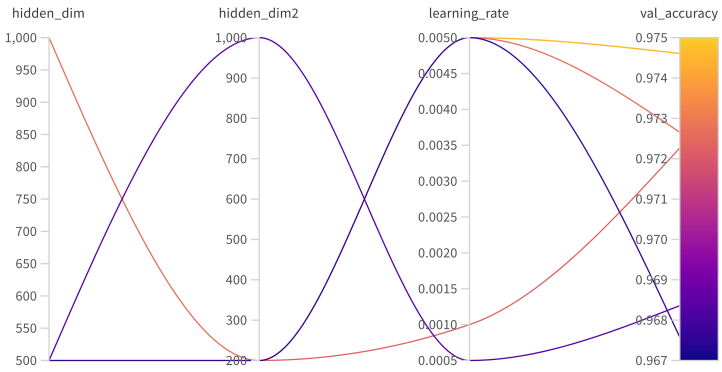
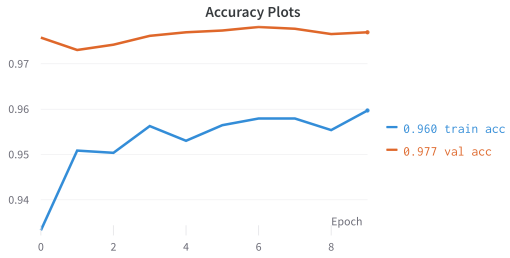


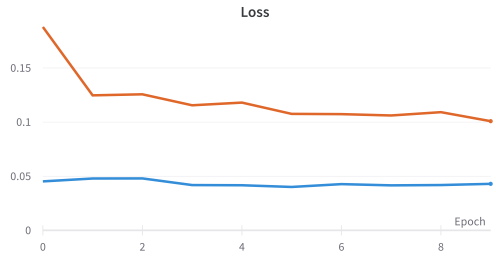
Figure 16: Parameter Tuning

We choose the best parameters from the above plot and train our final model. The training curves are in 17a and 17b

Observations: From the above plots and graphs we observe that hyperparmeter tunning



(a) Accuracy Plots



(b) Loss plots

had negligible effect on the validation accuracy. This may be true because we are using a pretrained model for a very simple task. We also note that the fine-tunning converges in less than 10 epochs.

3.1.2 Results

We plot the confusion matrix to get more interpretable result compared to accuracy.

Observation: We can observe that the classes *rango* and *farfalla* are the most confused

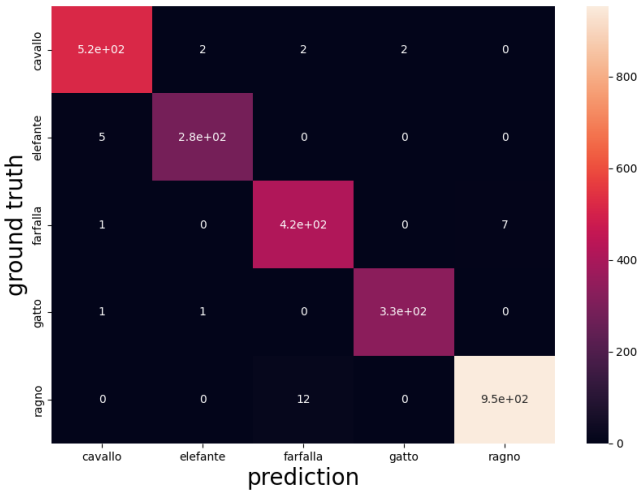


Figure 18: Confusion Matrix

having 12 false positives.

Based on these observations we analyse some images that have been mis-classified to see if the model was at fault or the data was noisy. Based on the 19a, 21b and 21a the mis-classified images are a mix of noisy data and model errors, a heavier weight on noisy data.

(a) Predicted Image: Elefante
Ground Truth: Cavallo

(b) Predicted Image: Farfalla
Ground Truth: Ragno

(a) Predicted Image: Cavallo
Ground Truth: Elefante

(b) Predicted Image: Elefante
Ground Truth: Gatto

(a) Predicted Image:Farfall
Ground Truth:Rango

(b) Predicted Image:Farfall
Ground Truth:rango

3.2 GoogleNet

We follow the similar process as mentioned before. We freeze the parameters of the GooglNet model and pass the dense CNN features to another trainable feed forward dense network with *ReLU* activation.

3.2.1 Training

The hyperparameters we could tune were only the batch size, learning rate and number of hidden nodes in the dense network. We observe that changing batch size had minimal effects on performance hence we decided to use a standard value of 64. We experiment with 4 learning rate values , while varying the other hyperparameters and observe the following.

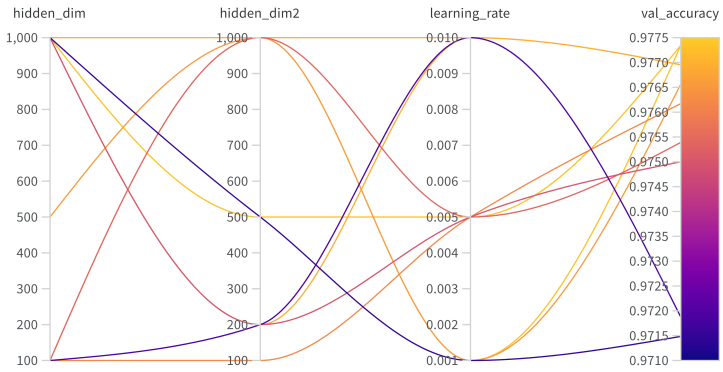
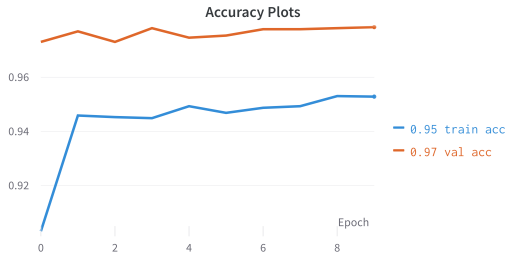


Figure 22: Parameter Tuning

We choose the best parameters from the above plot and train our final model. The training curves are in 23a and 23b. Again as seen before with the `VGG` model , the convergence takes place in less than 10 epochs.



(a) Accuracy Plots

(b) Loss plots

Results

We plot the confusion matrix to get more interpretable result compared to accuracy.

Observation: We analyse some images that have been miss-classified to see if the model

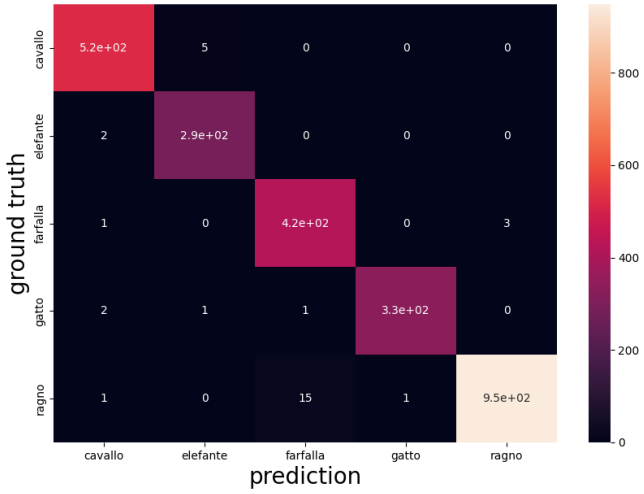


Figure 24: Confusion Matrix

was at fault or the data is noisy. We observe that `VGG` and `GooglNet` make similar mistakes and a major attribute is data noise as we can see in figures 27b or 27a are noisy images in which no clear animal is visible.

(a) Predicted Image: Elefante
Ground Truth: Cavallo

(b) Predicted Image: Farfalla
Ground Truth: Ragno

(a) Predicted Image: Cavallo
Ground Truth: Elefante

(b) Predicted Image: Elefante
Ground Truth: Gatto

(a) Predicted Image:Farfall
Ground Truth:Rango

(b) Predicted Image:Cavallo
Ground Truth:Farfalla

Comparision between VGG and GoogleNet

1. GooglNet has lesser parameters than VGG16 and yet achieves similar accuracies on out held out validation dataset, hinting to the fact that our image classification task

is not very complex.

2. Parameter tuning has a negligible effect as the deep CNN features have learned strong image representations.

4 Task 4

In addition to the previous setup, instead of using a pre-trained model, we implement a convolution network consisting of 2 convolution and 2 mean pooling layers. We pass the CNN features to a feed forward neural network in-order to classify the images.

Training

As we have several hyperparamters that are tunable , we explore several parameters for the number of feature maps for the last convolution layer along with number of hidden nodes in the feed forward layers. We obtain the following plot. In figure 28 we can see that the best parameter for number of feature maps for CL2 is 10.

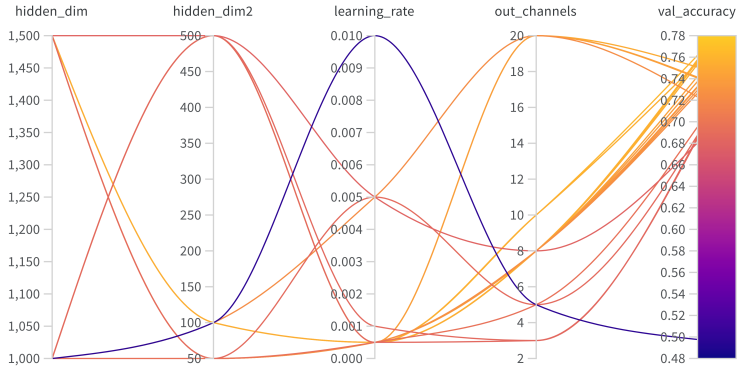
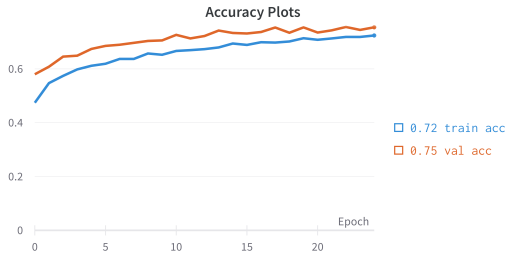
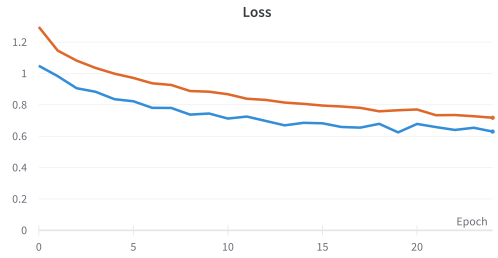


Figure 28: Parameter Tuning

We choose the best parameters from the above plot and train our final model. The training curves are in 29a and 29b. We obtain a final validation accuracy of 75 % . We also note that as no overfitting is taking place there is no need to use layers like Dropouts.



(a) Accuracy Plots



(b) Loss plots

4.0.1 Results

We plot the confusion matrix to get more interpretable result compared to accuracy.

Observation: Figure 30 shows a clear diagonal relationship as anticipated but we can note

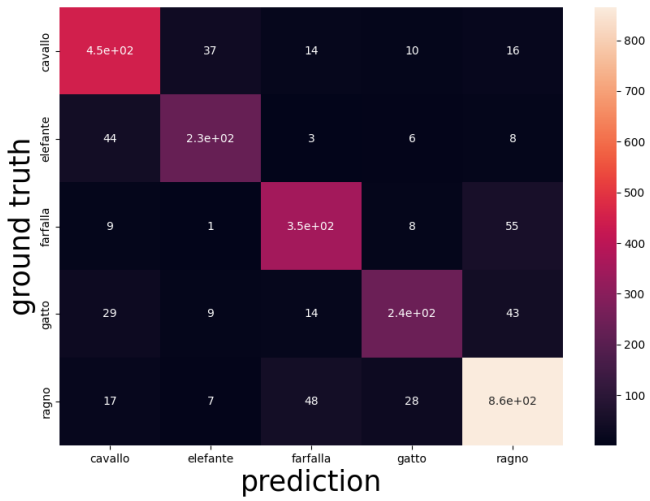


Figure 30: Confusion Matrix

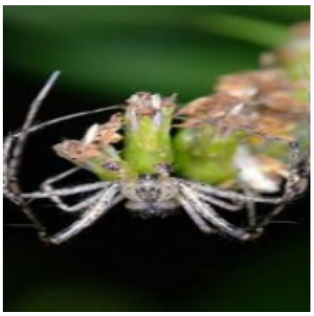
that some images have been wrongly classified. We now analyse these mis-classified images to see if the model was at fault or the data is noisy.

(a) Predicted Image:Elefante
Ground Truth:Cavallo

(b) Predicted Image:Elefante
Ground Truth:Cavallo

(a) Predicted Image:Gatto
Ground Truth:Rango

(b) Predicted Image:Rango
Ground Truth:Gatto



(a) Predicted Image:Farfall
Ground Truth:Rango

(b) Predicted Image:Cavallo
Ground Truth:Elefante

Observations:

- Here we can clearly see that , specially for the `Cavallo` and `Elefante` class that the model is making obvious mistakes. One possible hypothesis is that the model we have trained from scratch has learned some superficial features like the color grey is implies strong likelihood with the class `Elefante` as predominately the images from this class are greyish.
- We can also note that for the images where the entity is not very clear or distinct with respect to its backround, the model fails to recognise the true class
- Another clear observation based upon the accuracies and plots is that using a pre-trained model that has seen a diverse set of images always helps learn more robust features. Also model convergence takes longer when training a model from scratch.

5 Overall observations

1. We find than CNN are much superior models for image classification than the standard MLFFN model.

2. CNN's are able to learn local features in an image and hence give much better representation of images than normal global histogram vectors.

3. Local features help to represent each region of an image differently and hence the models can learn different intricacies in the image data that cannot be matched by just doing a global feature extraction.

4. We had observed in confusion matrices of task 1 and task 2 that the models had failed completely in differentiating the images of the 3 classes 'carvallo' , 'farfella' and 'ragno' and we had attributed this failure to the fact that the images in these classes have similar color schemes.

5. But from the confusion matrix of CNN models we find that CNN's are extremely successful in classifying the images of these 3 classes too. This is again because that

CNN's learn local features in the images too. Thus even though 2 images may have the same color scheme globally , each different region of the image may differ in the colors present in that region or in many local factors too which may not differ globally as a whole.

6. Thus we conclude that CNN's are extremely powerful in image classification and give superior results as compared to other traditional image classification models.