Statistical Pattern Recognition - Project 2

Souradip Chakraborty

Experimental Analysis on Dataset 1: MNIST

Dataset Description: The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

Reference : http://yann.lecun.com/exdb/mnist/

Dataset Used : data.mat

Task Description: The classification tasks is to identify hand-written digits

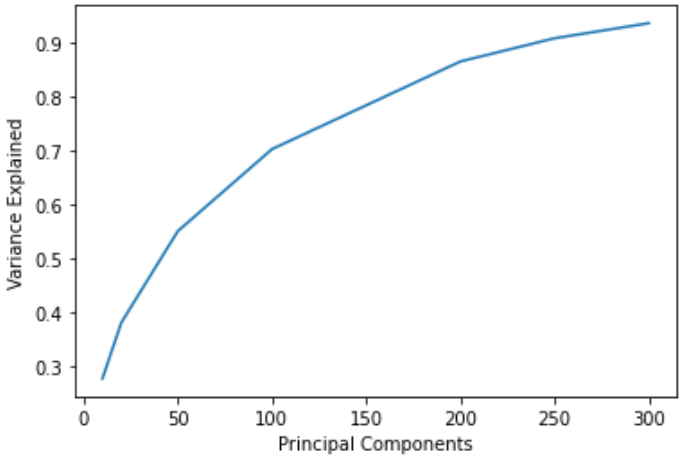
Tasks	No of Unique Labels	Train Size	Test Size
Image of Digits	10	60,000	10,000

Feature Pre-Processing :

1. **Principal Component Analysis :**

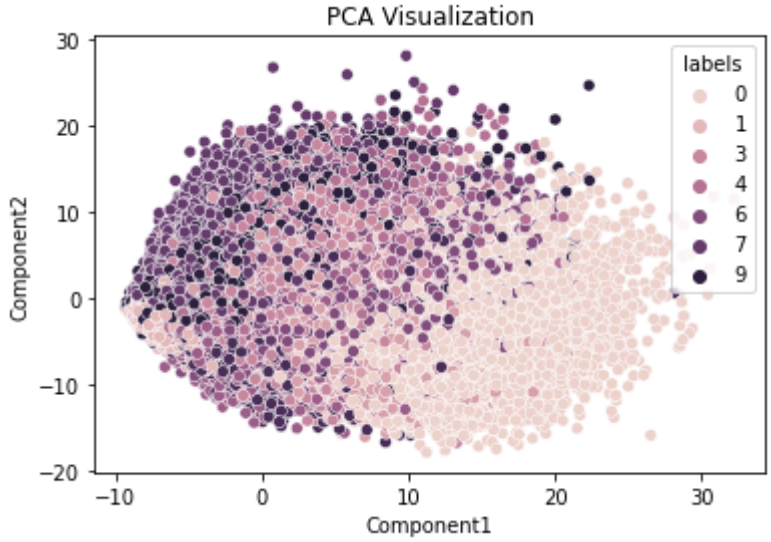
Here I apply Principal Component Analysis on the feature space and observe the Variance explained in the feature space by the Principal Component Analysis

It is important to note that I used Standardization of the data before applying Principal Component Analysis else it might be sensitive to the variance of the component.



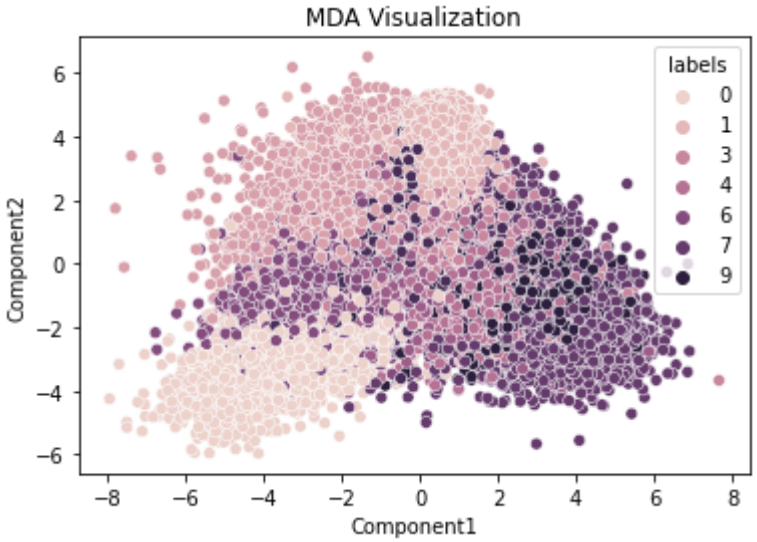
As it can be seen that with 250 components, we can explain 90% of the variance in the data. Hence, will be taking 250 components in Logistic regression. However, due to computation constraints in SVM we will restricting our analysis to **50 PCA components** which explains 55% of the variance in the data.

Here, I try to visualize the data distribution with 2 PC components to understand that how much it is able to capture the variability in Y. It can be seen even with 2 components, it can capture some aspects of the digits classification and getting some separability as clusters.



2. **Multiple Discrminant Analysis (MDA) :**

For this, the max number of components can be 9 since, the total no. of classes are 10 in MNIST and hence i take the no. of components as 9. Although this means its not a very fair comparison between MDA and PCA but it is restricted by the methodology.



It is very intuitive to visualize the data-distribution of MDA when compared to PCA and we can clearly see that MDA is producing a better separability of the data with 2 components.

Logistic Regression Analysis on MNIST

1. Logistic Regression with PCA

Here , I implement Logistic regression on the MNIST data with **250 PCA Components**

Now, i have created a 80-20 validation split from my training data to ensure and fine-tune on the hyperparameters, primarily the learning rate.

Finally , will be reporting the score on the three accuracy scores

Train Accuracy	93.22%
Validation Accuracy	92.55%
Test Accuracy	92.62%

2. Logistic Regression with MDA

As mentioned, here I implement Logistic regression with MDA/LDA with **9 LDA components** and report the accuracy performance on the three components : train, validation and test.

Train Accuracy	88.98%
Validation Accuracy	89.36%

Test Accuracy	88.67%

PCA outperforms MDA in the classification task as can be seen from the scores shown above.

Support Vector Machines Analysis on MNIST

In this section, I implement the various types of SVM including Linear, Polynomial and Kernel SVM with PCA and LDA on the MNIST Image classification task. It is important to note the following hyperparmaters are important for SVM

Hyperparameters

1. *C : regularization, default = 1*
2. *kernel = 'linear', 'poly', 'rbf'*
3. *degree = 3,4.. for 'poly'*
4. *Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.*
   *gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,*

1. Linear SVM with PCA

As mentioned before, PCA components are kept at 50 which explains 55% of the variance in data and apply linear SVM and report the three scores. The SVM implementation is slow and takes time when the number of features are very high

Linear SVM with PCA	
Train Accuracy	93.4 %
Validation Accuracy	93%
Test Accuracy	93.3%

It is very interesting to observe that even with 50 components, linear SVM is able to achieve high accuracy which indicates that the features which explains most of the variance in X also explains most of the variance in the labels space.

2. Polynomial SVM with PCA

For the polynomial kernel, I implement the same with r = 3 i.e polynomial degree.
The number of PCA components **are kept at 50**. However, it takes a lot of time to perform with Polynomial kernel.

Polynomial SVM with PCA	
Train Accuracy	98.1%

Validation Accuracy	97.3%
Test Accuracy	96.72%

3. RBF Kernel SVM with PCA

Here I have default value of gamma which is calculated based on *1 / (n_features * X.var())*

Kernel SVM with PCA	
Train Accuracy	98.82%
Validation Accuracy	90.48
Test Accuracy	90.47

4. Linear SVM with MDA

As mentioned here, I implement MDA with 9 components.

Linear SVM with MDA	
Train Accuracy	89.67%
Validation Accuracy	90.04%
Test Accuracy	89.37%

5. Polynomial SVM with MDA

As mentioned, I keep the r = 3 in polynomial SVM

Polynomial SVM with MDA	
Train Accuracy	93.22
Validation Accuracy	91.97
Test Accuracy	91.75

It is very interesting to observe that Polynomial SVM with 9 MDA components is able to achieve an accuracy of approximately 92%.

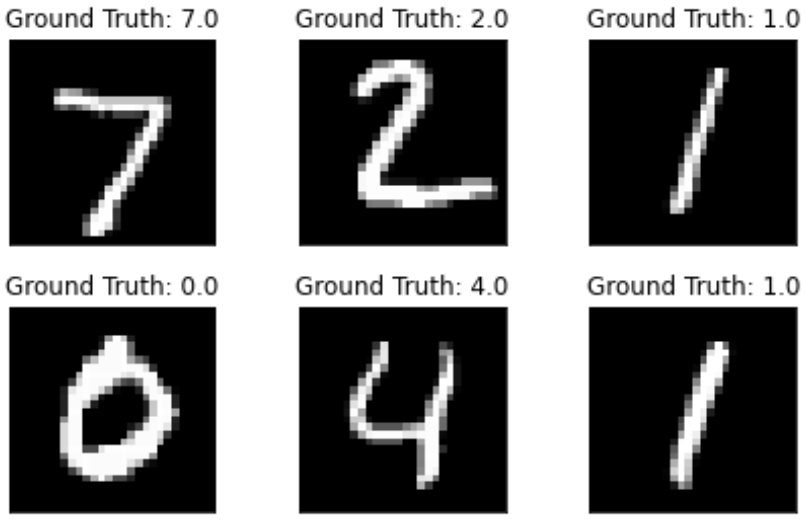
6. Kernel SVM with MDA

Kernel SVM with MDA	
Train Accuracy	94.11
Validation Accuracy	93.20
Test Accuracy	92.53

Although the comparisons made won't be exact and appropriate as the number of components are not exact. However, based on the current analysis it seems that Polynomial SVM with 50 PCA components gave a very high test accuracy of 96% performing best among these methods

Convolutional Neural Network Analysis on MNIST

In this section, I implement Deep Convolution Neural Network on the MNIST Classification task. I apply normaliztion before passing the image to the Convolution Neural Network model. As presented below, to check the sanity of the dataset I visualize some of the images with the labels as shown below.



For the CNN implementation the below architecture has been implemented

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

With a randomly initialized neural network, first I test the random model's performance on the test data and get an accuracy of 12% which is almost similar to random of 10% which makes sense.

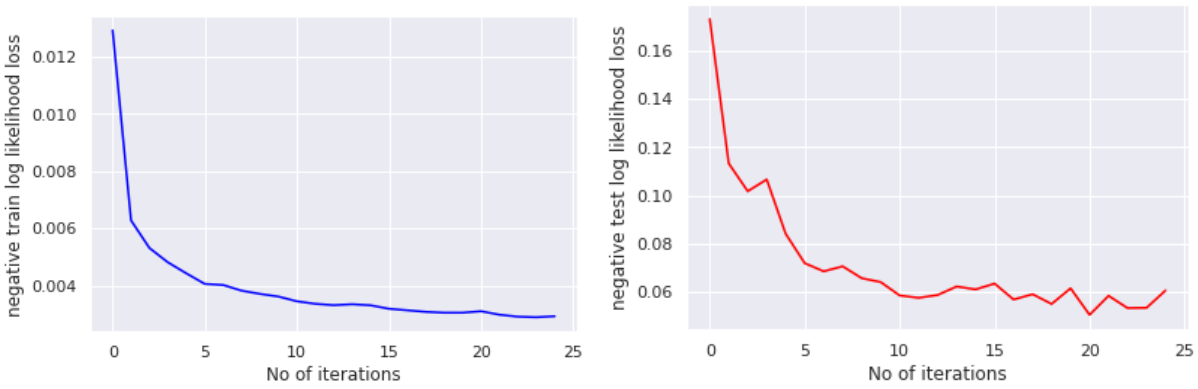
```
Test set: Avg. loss: 9.7785, Accuracy: 1179/10000 (12%)
```

Now, I train the CNN model with the following configuration and parameters :

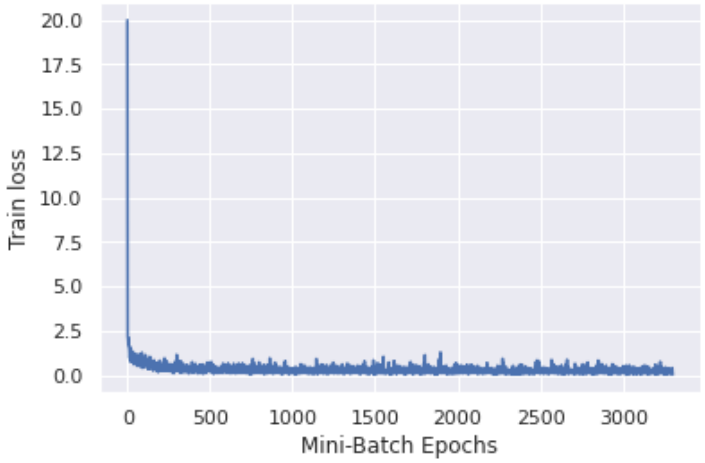
1. `n_epochs = 25`
2. `batch_size_train = 64`
3. `batch_size_test = 1000`
4. `learning_rate = 0.01`
5. `momentum = 0.5`
6. `log_interval = 10`

The model after training for 20 epochs is able to achieve a train accuracy of and a test accuracy of , which is highest amongst the options tried out.

The training and test loss decreases significantly in 25 epochs only as shown below :

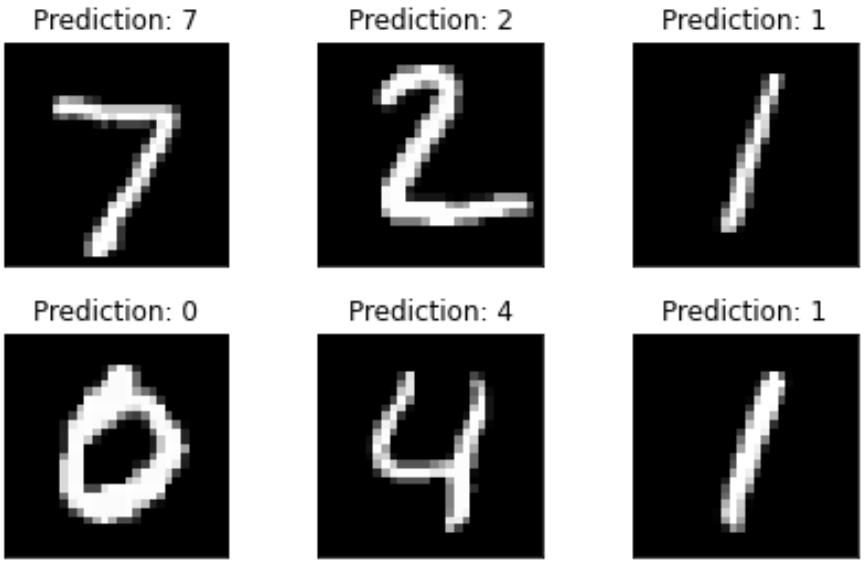


Now, from the above plot it might raise a question to why the train accuracy in the initial epochs are low. The reason is the plot is at an epoch level whereas training has been happening at minibatch and the number of minibatches are approx 60,000 / 64, which is high and model learn many times within an epoch as many gradient updates happen per min-batch. However, I have also visualized a plot considering the same below.: Here if you see, the initial train loss is high which is the true picture.



Deep CNN	
Train Accuracy	99.91%
Test Accuracy	99%

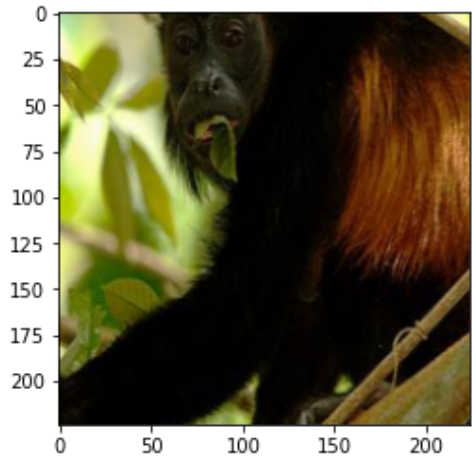
The predictions for few sample are shown below.



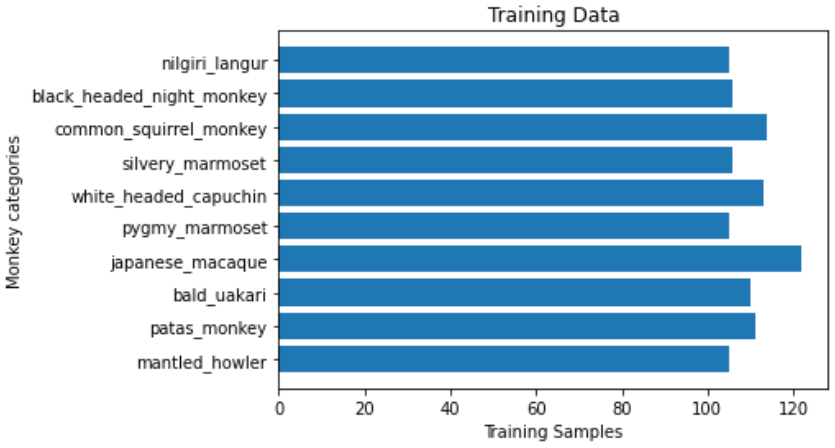
Convolutional Neural Network Analysis with and without Transfer Learning on Monkey Species Classification Data

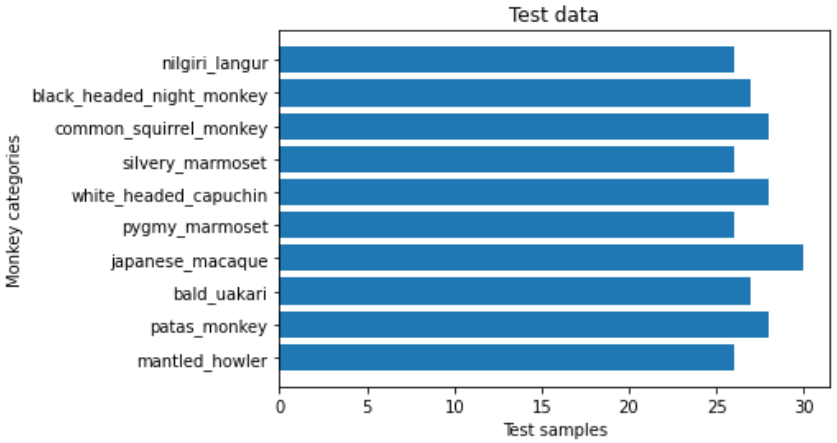
The dataset consists of two files, training and validation. Each folder contains 10 subforders labeled as n0~n9, each corresponding a species form Wikipedia's monkey cladogram. Images are 400x300 px or larger and JPEG format (almost 1400 images)

Label	Latin Name	Common Name	Train Images	Validation Images
n0	alouatta_palliata	mantled_howler	131	26
n1	erythrocebus_patas	patas_monkey	139	28
n2	cacajao_calvus	bald_uakari	137	27
n3	macaca_fuscata	japanese_macaque	152	30
n4	cebuella_pygmea	pygmy_marmoset	131	26
n5	cebus_capucinus	white_headed_capuchin	141	28
n6	mico_argentatus	silvery_marmoset	132	26
n7	saimiri_sciureus	common_squirrel_monkey	142	28
n8	aotus_nigriceps	black_headed_night_monkey	133	27
n9	trachypithecus_johnii	nilgiri_langur	132	26



Analyzing the Monkey Classification Dataset :





It is important to note that the training and test distribution across classes have not changed a lot and are similar which ensures the iid nature of the data to apply our CNN model which is ensured in this case.

Convolution Neural Network Details and Analysis

In this task, initially I will be fitting intially a Custom CNN model to classify the monkey species as mentioned above. For the task, I have implemented the following transformations

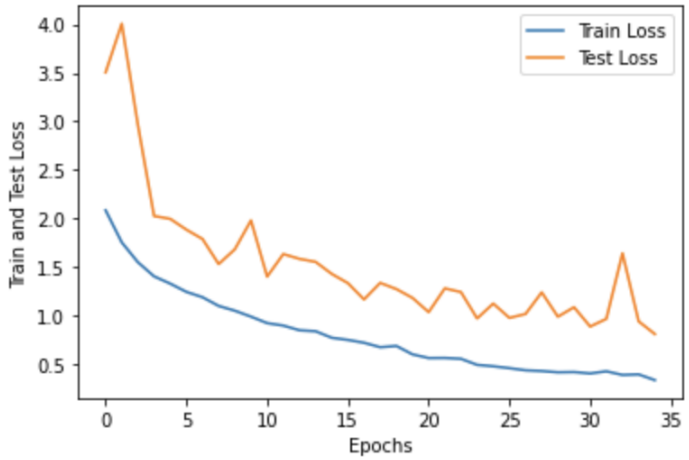
1. Resizing the Image
2. Center - Crop
3. RandomHorizontalFlip

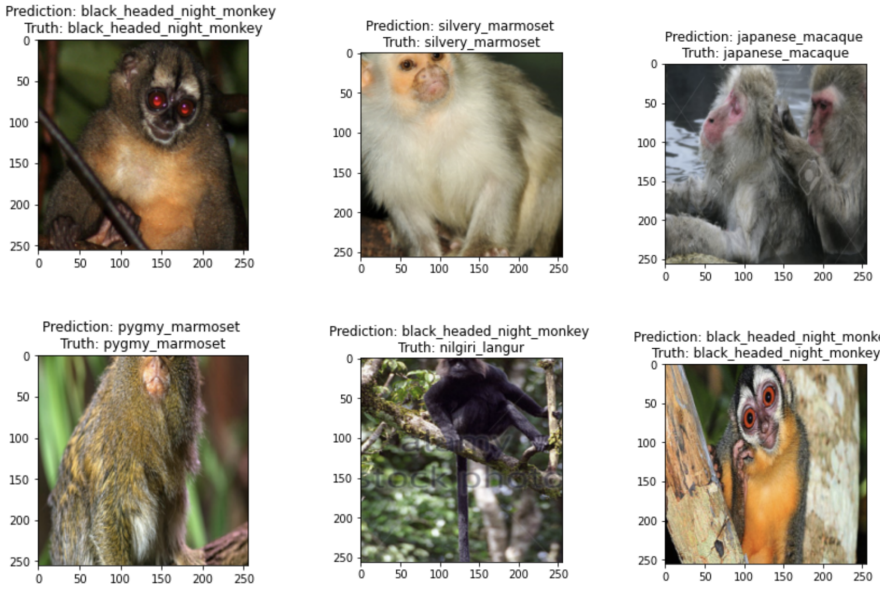
Now 3. Is done only for the images in the train set and not the test set and the main purpose is to provide a type of invariance i.e horizontal flipping doesn't change the classification. Invariances of these sort help in enhancing the robustness of the CNN classification model.

Now for the CNN architecture, I use the below architecture with 5 convolution layers, max pooling and Batch-Normalisation layers. The Batch-Normalisation layer is extremely important especially with RELU and Dropout. In this network, I have used ADAM Optimizer

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2))
  (batch_32): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2))
  (batch_64): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (batch_128): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (batch_256): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(256, 512, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (batch_512): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.45, inplace=False)
  (fc1): Linear(in_features=32768, out_features=64, bias=True)
  (batch_fc): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
```

The Hyperparameters used :
Epochs = 35
Learning Rate = 0.001
optimizer = Adam Optimizer
criterion = nn.CrossEntropyLoss()





The model achieved a test-accuracy of 76.12% which is not appropriate for the classification problem and gets a Train Loss: 0.3392, Test Loss: 0.8131. Now, we will try to analyze the performance with Pre-trained network i.e ResNet-18 which has been previously trained on ImageNet

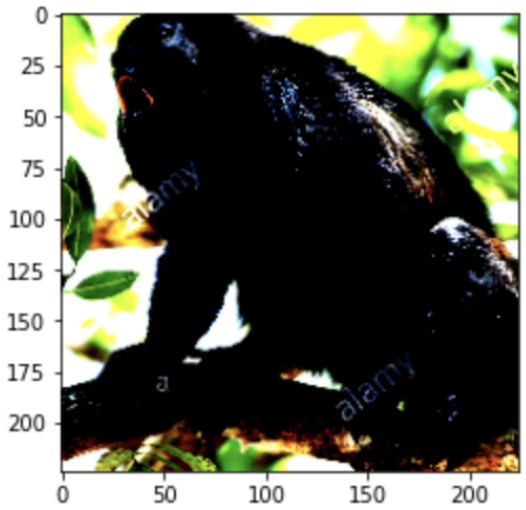
Transfer Learning with Pre-Trained Convolutional Neural Network :

In this section I will be implementing transfer learning with ResNet-18 and fine-tuning the last layers with task-specific information from the monkey datatset using Transfer Learning.

It is important to note that while doing Transfer Learning with any Imagenet model, we need to be consistent with the data transformation as done in there architecture.

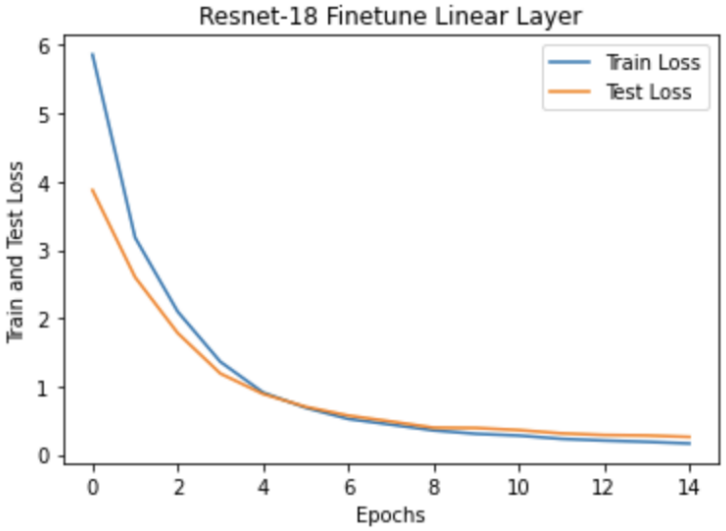
1. *Resize —> 256 * 256*
2. *Batch Size : 64*
3. *CenterCrop → 224 * 224*
4. *Normalize : mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]*

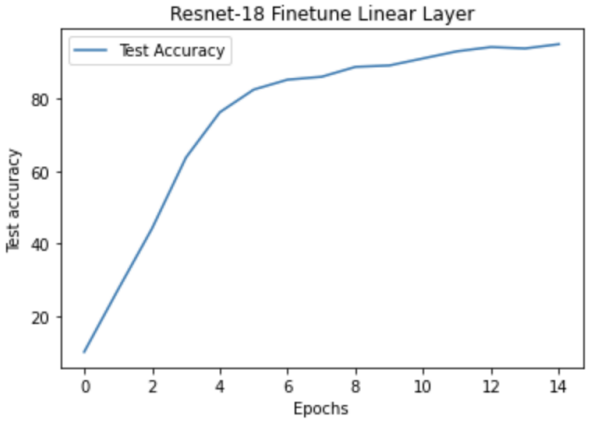
The above transformation is extremely important for meeting the consistency criterion discussed above.



First, I will modify the final layer in the Resnet architecture and use the method of tuning only the final linear of the Resnet architecture where I keep all other layer weights frozen as shown below.

```
layer4.0.bn2.weight
layer4.0.bn2.bias
layer4.0.downsample.0.weight
layer4.0.downsample.1.weight
layer4.0.downsample.1.bias
layer4.1.conv1.weight
layer4.1.bn1.weight
layer4.1.bn1.bias
layer4.1.conv2.weight
layer4.1.bn2.weight
layer4.1.bn2.bias
fc.weight
fc.bias
```

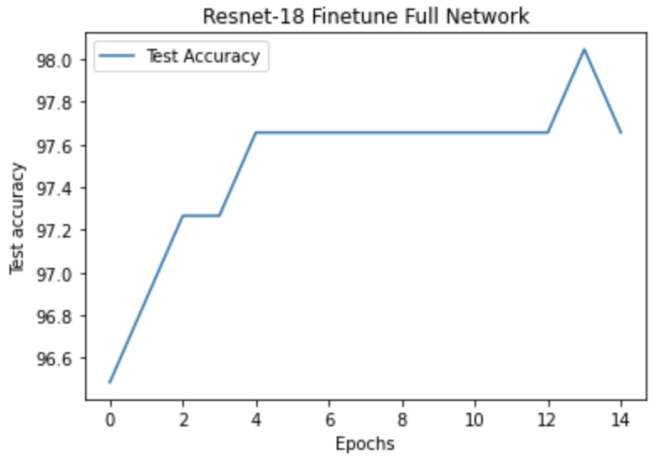





```
100%|██████████| 17/17 [00:50<00:00,  2.95s/it]
100%|██████████| 4/4 [00:13<00:00,  3.26s/it]
Epoch 9, Train Loss: 0.3534, Test Loss: 0.3935, Test Accuracy: 88.672
100%|██████████| 17/17 [00:48<00:00,  2.85s/it]
100%|██████████| 4/4 [00:14<00:00,  3.65s/it]
Epoch 10, Train Loss: 0.3039, Test Loss: 0.3892, Test Accuracy: 89.062
100%|██████████| 17/17 [00:49<00:00,  2.89s/it]
100%|██████████| 4/4 [00:13<00:00,  3.47s/it]
Epoch 11, Train Loss: 0.2783, Test Loss: 0.3584, Test Accuracy: 91.016
100%|██████████| 17/17 [00:48<00:00,  2.88s/it]
100%|██████████| 4/4 [00:13<00:00,  3.37s/it]
Epoch 12, Train Loss: 0.2301, Test Loss: 0.3104, Test Accuracy: 92.969
100%|██████████| 17/17 [00:49<00:00,  2.93s/it]
100%|██████████| 4/4 [00:14<00:00,  3.54s/it]
Epoch 13, Train Loss: 0.2052, Test Loss: 0.2866, Test Accuracy: 94.141
100%|██████████| 17/17 [00:48<00:00,  2.87s/it]
100%|██████████| 4/4 [00:14<00:00,  3.60s/it]
Epoch 14, Train Loss: 0.1862, Test Loss: 0.2795, Test Accuracy: 93.750
100%|██████████| 17/17 [00:49<00:00,  2.91s/it]
100%|██████████| 4/4 [00:12<00:00,  3.17s/it]
Epoch 15, Train Loss: 0.1624, Test Loss: 0.2581, Test Accuracy: 94.922
```

The train ,test accuracy and loss plots clearly indicates that fine-tuning the neural network has caused a significant increase in the test accuracy of the model and it reached 95% in 15 epochs whereas the original custom CNN model could only reach till 75% test accuracy with 35 epochs. This shows that pre-trained models capture a lot of inbuilt information while being trained on a huge dataset like ImageNet.

Finally, I fine-tune on the whole network to get the final accuracy boost but with an extremely low learning rate of 0.00001 else it might deviate and start training the initial layers which is not expected. It can be clearly seen from the below figures that the test accuracy reached a maxima of 98% with 15 epochs of training which is the highest amongst the possible combinations.




```
Epoch 7, Train Loss: 0.0123, Test Loss: 0.1177, Test Accuracy: 97.656
100%|██████████| 17/17 [00:48<00:00,  2.87s/it]
100%|██████████| 4/4 [00:13<00:00,  3.48s/it]
Epoch 8, Train Loss: 0.0099, Test Loss: 0.1145, Test Accuracy: 97.656
100%|██████████| 17/17 [00:49<00:00,  2.92s/it]
100%|██████████| 4/4 [00:13<00:00,  3.49s/it]
Epoch 9, Train Loss: 0.0084, Test Loss: 0.1121, Test Accuracy: 97.656
100%|██████████| 17/17 [00:50<00:00,  2.95s/it]
100%|██████████| 4/4 [00:14<00:00,  3.60s/it]
Epoch 10, Train Loss: 0.0071, Test Loss: 0.1113, Test Accuracy: 97.656
100%|██████████| 17/17 [00:49<00:00,  2.92s/it]
100%|██████████| 4/4 [00:13<00:00,  3.30s/it]
Epoch 11, Train Loss: 0.0069, Test Loss: 0.1079, Test Accuracy: 97.656
100%|██████████| 17/17 [00:49<00:00,  2.90s/it]
100%|██████████| 4/4 [00:11<00:00,  2.99s/it]
Epoch 12, Train Loss: 0.0058, Test Loss: 0.1070, Test Accuracy: 97.656
100%|██████████| 17/17 [00:51<00:00,  3.00s/it]
100%|██████████| 4/4 [00:13<00:00,  3.34s/it]
Epoch 13, Train Loss: 0.0056, Test Loss: 0.1018, Test Accuracy: 97.656
100%|██████████| 17/17 [00:50<00:00,  2.99s/it]
100%|██████████| 4/4 [00:13<00:00,  3.30s/it]
Epoch 14, Train Loss: 0.0049, Test Loss: 0.0806, Test Accuracy: 98.047
100%|██████████| 17/17 [00:49<00:00,  2.89s/it]
100%|██████████| 4/4 [00:13<00:00,  3.27s/it]
Epoch 15, Train Loss: 0.0045, Test Loss: 0.1010, Test Accuracy: 97.656
```

Discussion and Conclusion

It can be clearly observed that the CNNs gave the highest accuracy in both the test and outperformed all the other models. However, it is important to note that even Kernel SVM methods can be designed to achieve good accuracy as can be seen from the experiments above but is extremely expensive and slow to achieve that and the complexity increases with the no of dimensions as well. Deep CNN on the other hand are extremely fast and efficient using GPU based computations and was able to capture complex relations much faster. The most interesting part of the experiment was the fact that Transfer Learning worked very nicely and fine-tuning the Res-Net models gave 98% test accuracy which is excellent. This highlights the fact that Pre-trained models are extremely informative and can be used to extract features/embedding or fine-tuning for downstream tasks.