UNIVERSITY OF MARYLAND, COLLEGE PARK

Classification using SVM, CNN and Transfer Learning

Pranav Inani

pranavinani94@gmail.com

December 2018

Contents

1 Hand-Written Digit Recognition

The MNIST dataset containing 60,000 training images and 10,000 test images of handwritten digits was used to perform Hand-written Digit Recognition. Hence, this was a multi-class classifcation problem with 10 classes viz., the digits 0-9. Each image is normalized and centered in a 28x28 field. Hence, no post processing was required. Dimensionality reduction techniques were not used since the data is fairly low dimensional and training was done on a powerful machine with 8 cores and a GPU.

The classification was first done using SVM with different Kernels and then using a convolutional Neural Network. Finally their results were summarized.

1.1 Classification of MNIST dataset using SVM

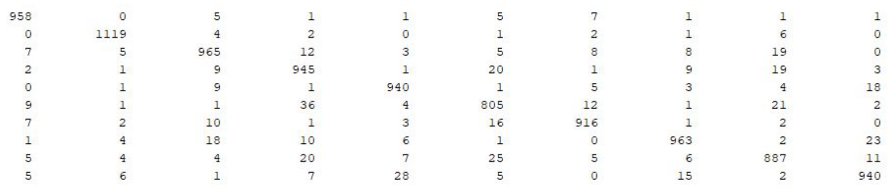
The classification problem described above was performed with Linear SVM, SVM with polynomial Kernel of orders 2 and 3, and with RBF kernel.

1.1.1 Linear SVM

Multiclass SVM was performed using MATALB's **fitcecoc()** function which is an error-correcting output codes (ECOC) model that uses linear SVM as default. The mis-classification cost **C** was set to 10. The algorithm performs **one vs one** classification such that for each binary learner, one class is positive, another is negative, and the software ignores the rest. This design exhausts all combinations of class pair assignments.

The accuracy achieved using Linear SVM: **94.38**

The confusion matrix obtained is shown in fig. 1.1



958	0	5	1	1	5	7	1	1	1
0	1119	4	2	0	1	2	1	6	0
7	5	965	12	3	5	8	8	19	0
2	1	9	945	1	20	1	9	19	3
0	1	9	1	940	1	5	3	4	18
9	1	1	36	4	805	12	1	21	2
7	2	10	1	3	16	916	1	2	0
1	4	18	10	6	1	0	963	2	23
5	4	4	20	7	25	5	6	887	11
5	6	1	7	28	5	0	15	2	940

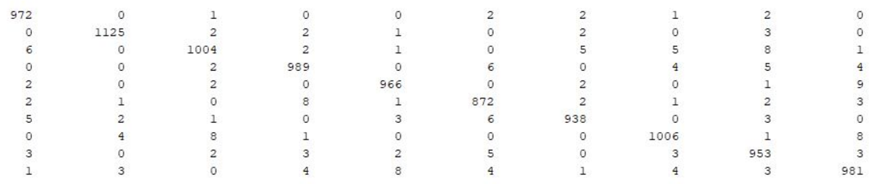
Figure 1.1: Confusion matrix of Linear SVM

1.1.2 SVM: Polynomial Kernel

Again MATALB's **fitcecoc()** function was used and a polynomial kernel template of order 2 and 3 was passed.

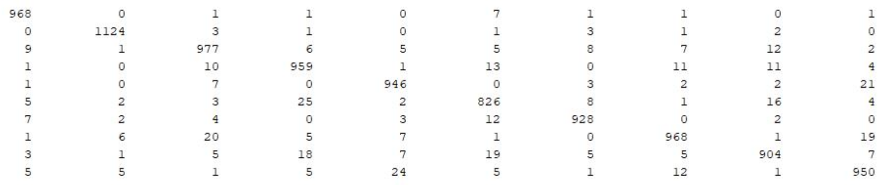
The accuracy achieved using Polynomial kernel of order 2 was: **98.06**
The accuracy achieved using Polynomial Kernel of order 3 was: **95.5**

The confusion matrix obtained for these kernels are show in in fig 1.2 and fig 1.3



972	0	1	0	0	2	2	1	2	0
0	1125	2	2	1	0	2	0	3	0
6	0	1004	2	1	0	5	5	8	1
0	0	2	989	0	6	0	4	5	4
2	0	2	0	966	0	2	0	1	9
2	1	0	8	1	872	2	1	2	3
5	2	1	0	3	6	938	0	3	0
0	4	8	1	0	0	0	1006	1	8
3	0	2	3	2	5	0	3	953	3
1	3	0	4	8	4	1	4	3	981

Figure 1.2: Confusion matrix: Polynomial SVM of order 2



968	0	1	1	0	7	1	1	0	1
0	1124	3	1	0	1	3	1	2	0
9	1	977	6	5	5	8	7	12	2
1	0	10	959	1	13	0	11	11	4
1	0	7	0	946	0	3	2	2	21
5	2	3	25	2	826	8	1	16	4
7	2	4	0	3	12	928	0	2	0
1	6	20	5	7	1	0	968	1	19
3	1	5	18	7	19	5	5	904	7
5	5	1	5	24	5	1	12	1	950

Figure 1.3: Confusion matrix: Polynomial SVM of order 3

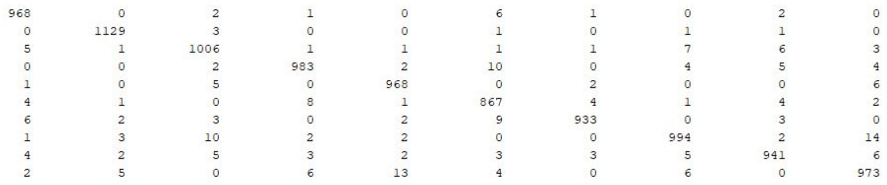
1.1.3 SVM: RBF Kernel

Strangely, for the RBF kernel the MATLAB **fitcecoc()** seemed to over-fitting on the traning data too much and was giving very poor testing performance. So, the RBF kernel results were produced using the LIBSVM library.

The radial basis kernel was used with $\gamma = 0.1$ and the cost for misclassification, **C** was set to 100.

The accuracy achieved using RBF kernel was: **97.62**

The confusion matrix obtained is shown in fig. 1.4



968	0	2	1	0	6	1	0	2	0
0	1129	3	0	0	1	0	1	1	0
5	1	1006	1	1	1	1	7	6	3
0	0	2	983	2	10	0	4	5	4
1	0	5	0	968	0	2	0	0	6
4	1	0	8	1	867	4	1	4	2
6	2	3	0	2	9	933	0	3	0
1	3	10	2	2	0	0	994	2	14
4	2	5	3	2	3	3	5	941	6
2	5	0	6	13	4	0	6	0	973

Figure 1.4: Confusion matrix of RBF kernel with $\gamma = 0.1$

1.2 Classification of MNIST dataset using CNN

The same classification task was performed using convolutional neural networks. The architecture used was similar to the one described in LeCun's paper [1]. The architecture, implemented in PyTorch, is briefly described as follows:
1) The first layer takes in images of input and applies 6 filters of kernel size 5x5 with a stride of 1. The input image is is augmented with padding of 2 so that the size of the image doesn't change. Then max pooling is applied with a 2x2 kernel with a stride of 2.
2) The output from the above step is input to another "conv-pool" layer similar to one described above, but this time with 16 kernel filters and no padding.
3) Next, the output from the above step is flattened into a vector of size $16 * 5 * 5 = 400$ and is

fully connected to a hidden layer with 120 nodes.
4) Then, another fully connected hidden layer with 84 nodes is added.
5) Finally, the above layer is fully connected with the output layer with 10 nodes, one for each class.

The activation function used in all the layers except the output is ReLU. The output layer uses the softmax activation function. The cost-criterion used was natural log-likelihood loss (NLL) and the optimizer used was stochastic gradient descent (SGD). The network was run for 5 epochs with a learning rate of 0.01 and momentum of 0.5 with a batch size of 64.

Fig 1.5 shows some example classifications performed by lenet5.

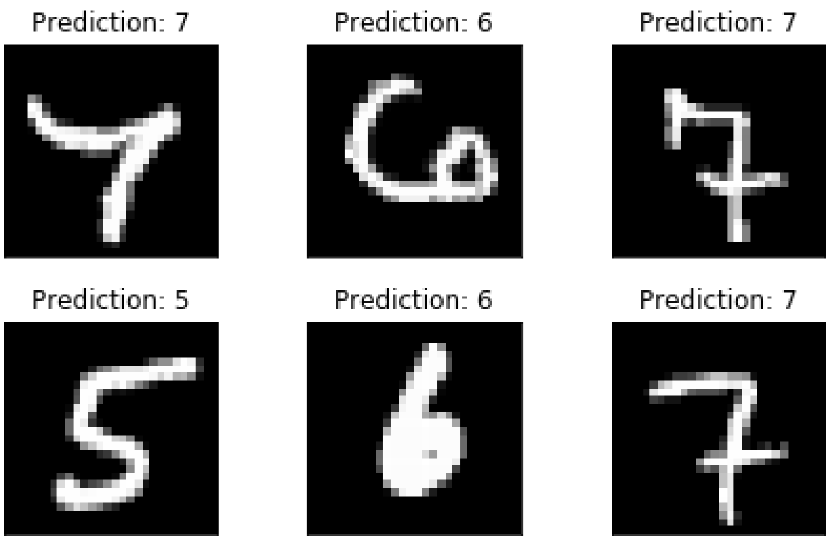
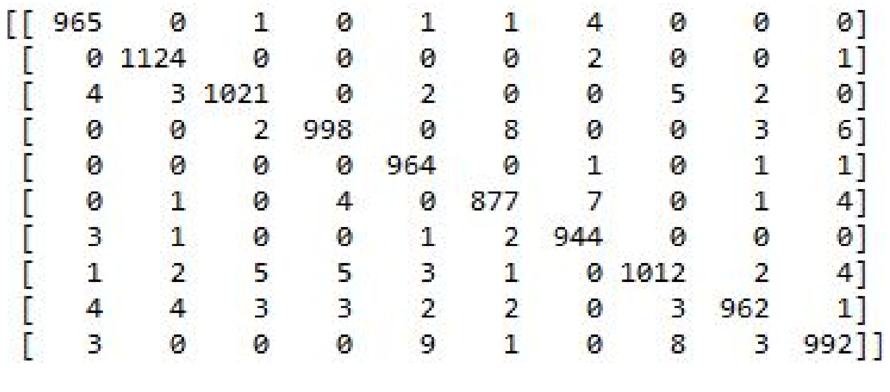


Figure 1.5: Confusion matrix with lenet5

The accuracy achieved using Lenet5 was: **98.59**
Fig 1.6 shows the confusion matrix for the CNN.



```
[[ 965    0    1    0    1    1    4    0    0    0]
 [   0 1124    0    0    0    0    2    0    0    1]
 [   4    3 1021    0    2    0    0    5    2    0]
 [   0    0    2  998    0    8    0    0    3    6]
 [   0    0    0    0  964    0    1    0    1    1]
 [   0    1    0    4    0  877    7    0    1    4]
 [   3    1    0    0    1    2  944    0    0    0]
 [   1    2    5    5    3    1    0 1012    2    4]
 [   4    4    3    3    2    2    0    3  962    1]
 [   3    0    0    0    9    1    0    8    3  992]]
```

Figure 1.6: Confusion matrix with lenet5

The loss curve clearly decreases over time and the test-error decreases along with it, suggesting that there aren't any major signs of overfitting. This can be seen in fig. 1.7

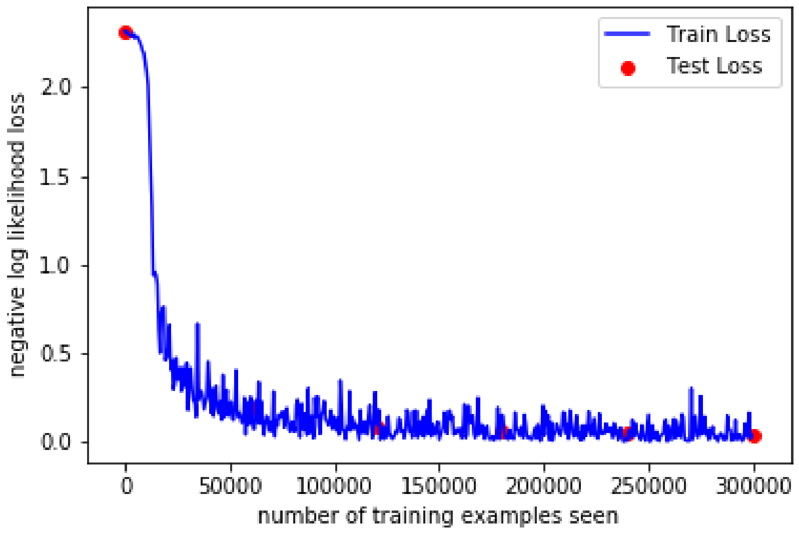


Figure 1.7: Test and train error vs # training examples

1.3 Results

The results clearly showcases the superiority of CNNs over SVM methods. While, the accuracy validation accuracy of polynomial kernel of order 2 (98.06%) comes close, the CNN with Lenet5 achieves an accuracy of 98.59% after just 5 epochs (about 2mins 30sec) while the SVMs took a lot longer to train.

However, it should be noted that all the SVM methods Linear(94.38%), Polynomial Kernel of order 3 (95.5%), Polynomial Kernel of order 2(98.06%) and RBF (97.62%) all had acceptable accuracy as well.

2 Monkey Species Classification Using CNN and Transfer Learning

2.1 Dataset

The 10-monkey species dataset [2] contains 1098 training images of 10 different monkey species and 272 validation images. An example image of each species of monkey can be seen in fig. 2.1 and the the names and some other miscellaneous data of the monkeys can be seen in fig 2.2.

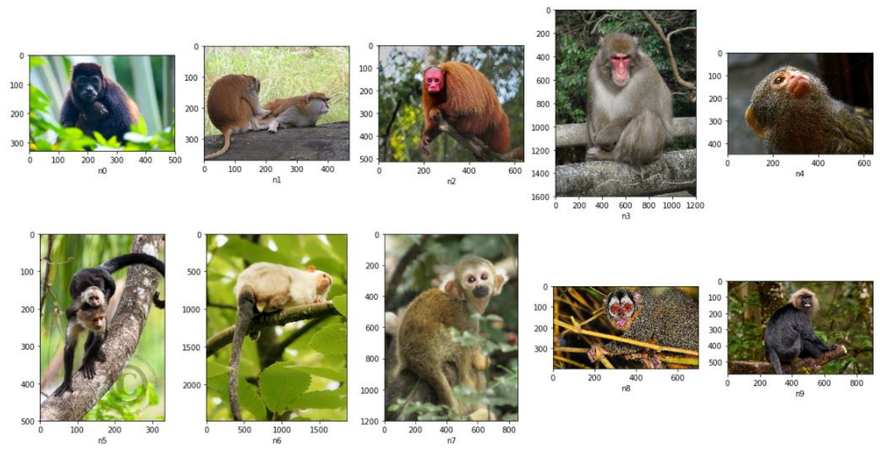
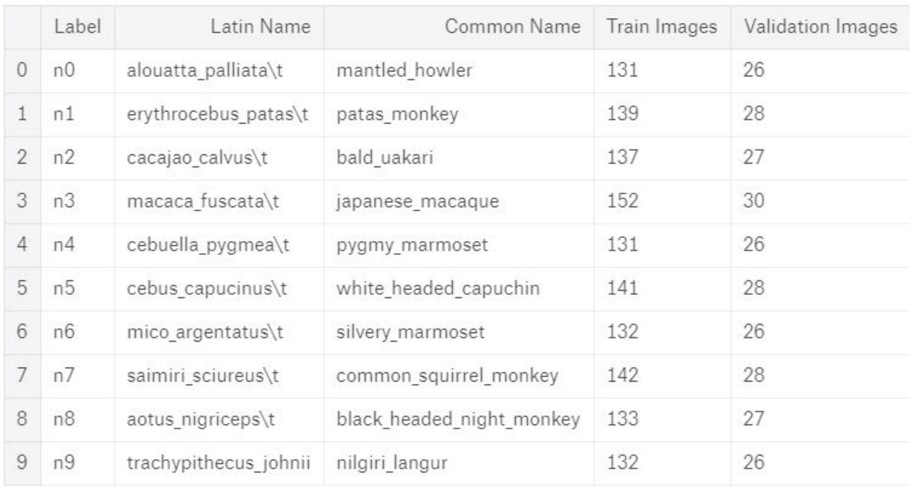


Figure 2.1: Examples Images of each Monkey Species



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

Figure 2.2: Monkey Data

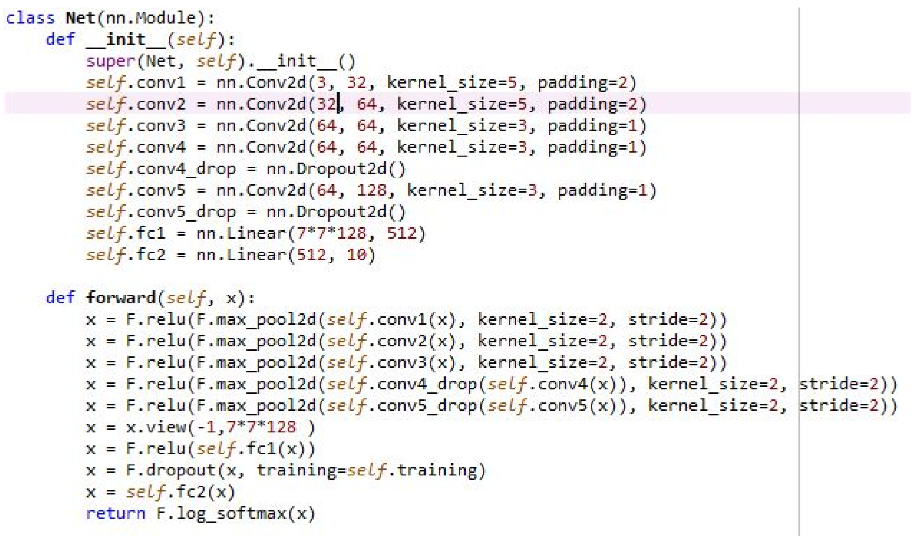
Since, images were of varying sizes, all the images were re-sized to 224 x 224 which is the input size to the Resnet18 model. Even the custom CNN model was re-sized to 224x224 to aid with fair comparison.

2.2 Custom CNN

To establish a baseline, first, a custom 5 layer deep convolutional neural net was built in PyTorch to classify the monkey data. The architecture consisted of 5 conv-pool layers as described in the previous sections. starting with 32 filters in the first layer and growing all the way to 128 filters at the output of the 5th conv-pool layer. While the input image at first layer is 224x224 and at the output it's 14x14. The first 2 layers had a kernel size of 5x5 and the next 3 layers had kernel size of 3x3. All the layers employed padding such that the size of the image did not change after applying the Kernel.

After the 5th conv-pool layer the tensor was unrolled and fully connected to a hidden layer with 512 nodes. Which was in-turn connected to the output layer. The activation function at all the layers except the output was ReLU. At output, softmax was used. The cost-criterion was Natural log-likelihood and the optimizer used was Adam.

Dropout layers were implemented in the last two convolution layers and the hidden layer. These promoted regularization and improved validation accuracy. The code snipped of the exact architecture can be seen in fig 2.3



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv4_drop = nn.Dropout2d()
        self.conv5 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv5_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(7*7*128, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv3(x), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv4_drop(self.conv4(x)), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv5_drop(self.conv5(x)), kernel_size=2, stride=2))
        x = x.view(-1,7*7*128 )
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

Figure 2.3: Architecture of the custom CNN

However, this the problem of over-fitting still persisted. So, L2 regularization or weight decay was used to get better validation accuracy. Figure 2.4 and fig 2.5 show the improvement in performance by adding a weight decay of $\gamma = 0.01$. Fig. [**?**] shows that the performance worsens on reduce the weight decay to $\gamma = 0.001$.

The training was done over 200 epochs and took about an hour to train with GPU.

The best accuracy (which corresponds to plot in fig 2.5 is: **75%**

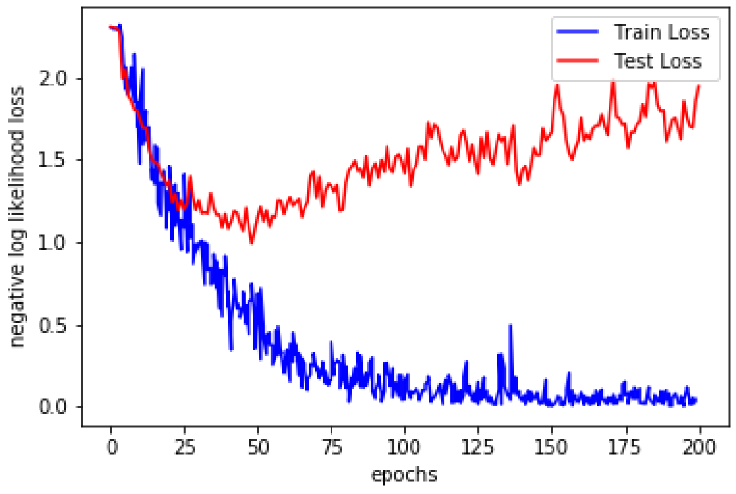


Figure 2.4: Without weight Decay

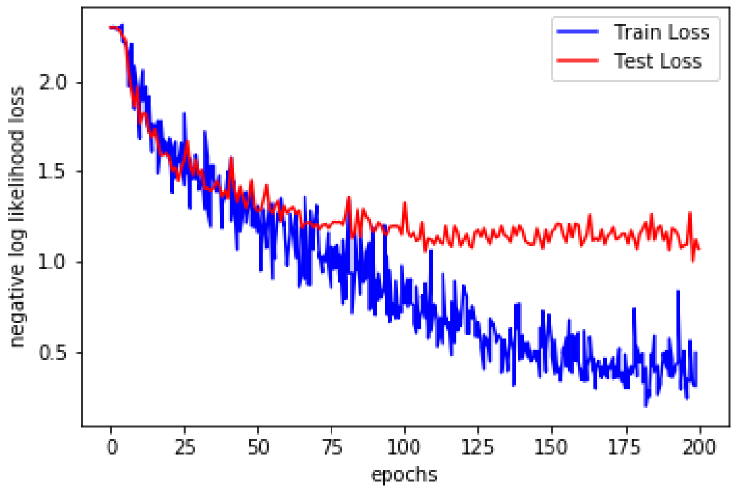


Figure 2.5: weight_decay = 0.01

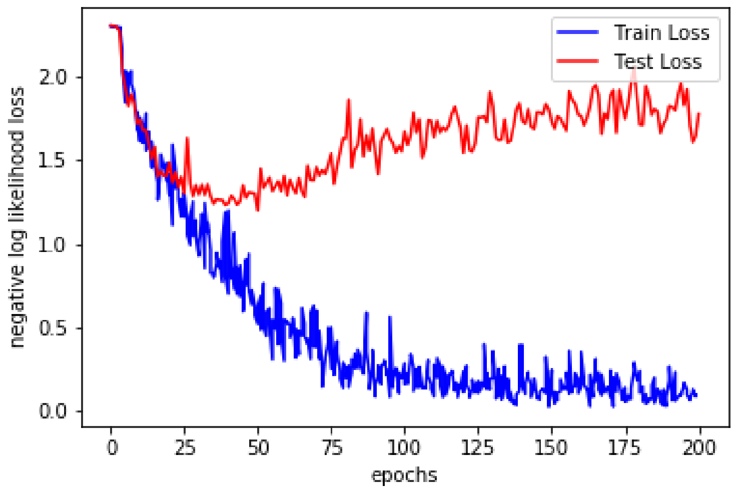


Figure 2.6: weight_decay=0.001

2.3 Transfer Learning with ResNet18 as feature Extractor

To implement transfer learning by using a pre-trained model as a feature extractor, a pre-trained ResNet18 model was downloaded and used as a feature extractor. The last layer (output layer) of ResNet18 was replaced with a output layer having 10 output nodes with linear activation function. The learning rate used was 0.001 and the traning was done for about 14 epochs. The loss-critereon was cross entropy loss and the optimizer was SGD. During training, all the convolutional layer parameters were frozen and the learning was only done at the fully connected layers. Hence, the convolutional layers act as generalized feature extractors. The loss vs epoch plot can be seen in fig 2.7

Validation accuracy for Transfer Learning with ResNet as Feature Extractor: **96.3235%**

This is already a massive improvement over the custom CNN trained in the previous section. The training time for this model was **0m 49s**. Which is incredibly fast! This is because only the fully connected layers at the end are being learned.

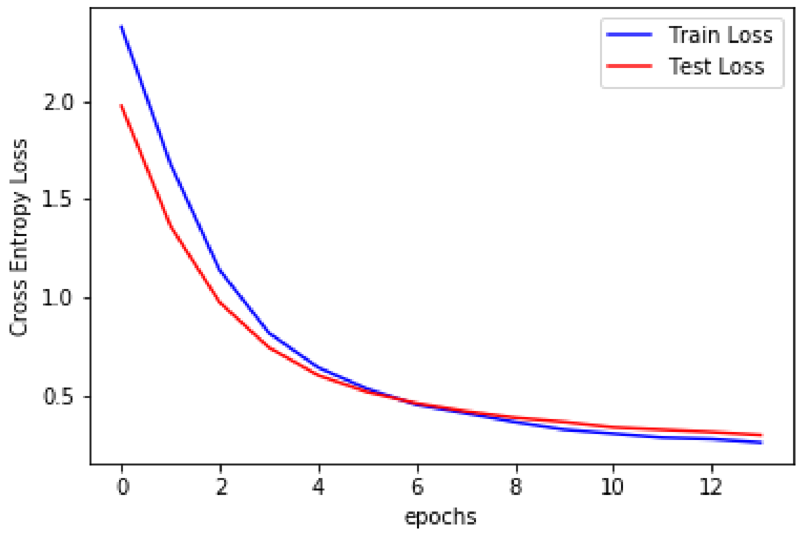


Figure 2.7: Loss vs epoch for transfer learning without fine tuning

2.4 Transfer Learning by fine tuning ResNet18

Finally, to perform transfer learning with fine turning, all the steps as described in the previous section were performed. However, this time the convolutional layers were not frozen and learning was allowed to be done throughout the network. The model was again run for 14 epochs with identical conditions as the previous experiment. It took about **3m 30s** to train. The best validation accuracy for Transfer Learning with fine tuning: **98.5294**

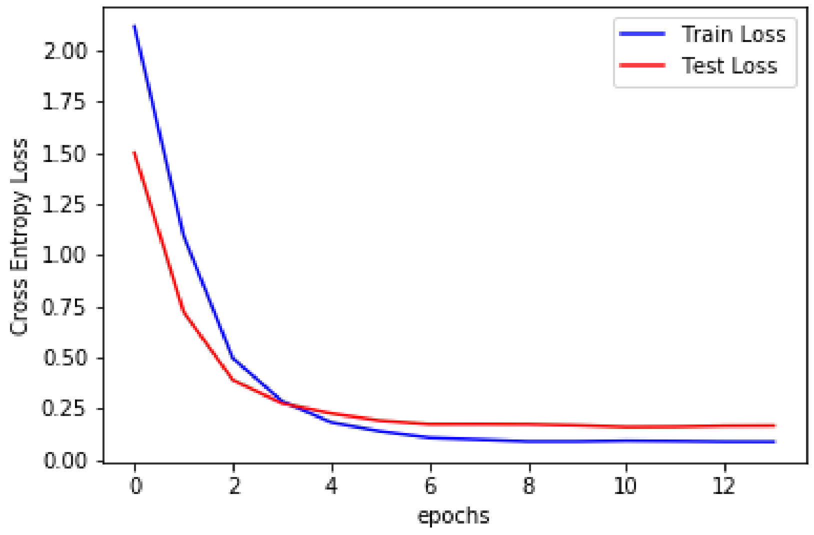


Figure 2.8: Loss vs epoch for transfer learning fine tuning

2.5 Results

Clearly, when large amounts of data is unavailable, CNNs are not as effective. This is evidenced by the poor validation accuracy of about **75%** Moreover, constructing custom networks is a painstaking task with a lot of hyper-parameters like kernel size, strides, number of layers proving to be quite a handful of design parameters to tweak and tune. Also, they take a really long time to train.

Hence, transfer learning turns out to be a handy tool to have. With the massive jump in accuracy to 96.32% without fine tuning and 98.52% with fine tuning. And they are also orders of magnitude faster to train with the feature extrator version being trained in less than a minute on GPU and the fine tuning model taking less than 5 minutes.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] "10 monkey species." `https://www.kaggle.com/slothkong/10-monkey-species`. Accessed: 2018-12-15.