University of Maryland, College Park

ENEE633/CMSC828C: STATISTICAL PATTERN RECOGNITION

Project2

Fall 2022

Kavyashree Devadiga UID. 117398045

INDEX

List of Figures

Chapter 1

Part I: Hand-written Digit Recognition

For handwritten recognition, the dataset used is MNIST. It was downloaded locally once and loaded for each of the following tasks. One other way to load the data without manually downloading the files is to use the Keras datasets load data method. The dataset comes with separate train and test folders so data splitting was not required. As a preprocessing step, the images were **standardised** using StandardScaler() where the standard score is calculated using as $(x - mean)/standarddeviation$ for every data point x in the training set. The transformer first fits the training data and transforms both training and testing data before feeding to the model. This was implemented using sklearn.preprocessing library method.

1.1 SVM

Implements linear and kernel SVM on MNIST dataset. You have to try different kernels (linear, polynomial, RBF) and compare results in your report.

Implementation :

Here the training dataset containing 60,000 images were randomly sampled and and a subset of 20,000 images were used to train. The complete MNIST test dataset was used for testing the model. Class-wise count of images can be seen in table 1.1

Table 1.1: Class-wise image count in datasets

Class	0	1	2	3	4	5	6	7	8	9
Train	2262	2053	2025	1988	1986	1968	1964	1962	1952	1840
Test	1135	1032	1028	1010	1009	982	980	974	958	892

1.1.1 Evaluation criteria

The model was fit on train data and a classification report was generated. This helps evaluate how well the model fits the training data. This model was then evaluated against unseen, test data and the classification report generated gave an idea of how well the model was able to generalize the given problem.

One example of the entire classification report for Linear SVM with PCA case can be seen in the tables 1.2 and 1.3 In some cases, even though the training metrics are good, the test accuracy does not match the results. This can be thought of as a slight overfit scenario. **The test accuracy is considered the true accuracy of the model.**

Table 1.2: Linear SVM with PCA Training Results

Class	precision	recall	f1-score	support
0	0.99	1.00	0.99	1984
1	0.98	0.99	0.98	2206
2	0.96	0.96	0.96	2000
3	0.93	0.94	0.93	2092
4	0.97	0.97	0.97	1976
5	0.95	0.94	0.94	1784
6	0.99	0.99	0.99	1995
7	0.97	0.96	0.96	2095
8	0.96	0.93	0.95	1903
9	0.95	0.94	0.94	1965
accuracy			**0.96**	20000
macro avg	0.96	0.96	**0.96**	20000
weighted avg	0.96	0.96	0.96	20000

Table 1.3: Linear SVM with PCA Testing Results

Class	precision	recall	f1-score	support
0	0.97	0.97	0.97	980
1	0.98	0.99	0.99	1135
2	0.95	0.92	0.93	1032
3	0.94	0.93	0.94	1010
4	0.96	0.95	0.96	982
5	0.95	0.93	0.94	892
6	0.93	0.95	0.94	958
7	0.85	0.95	0.90	1028
8	0.96	0.93	0.94	974
9	0.96	0.92	0.94	1009
accuracy			**0.95**	10000
macro avg	0.95	0.94	0.95	10000
weighted avg	0.95	0.95	0.95	10000

1.1.2 Other Permutations

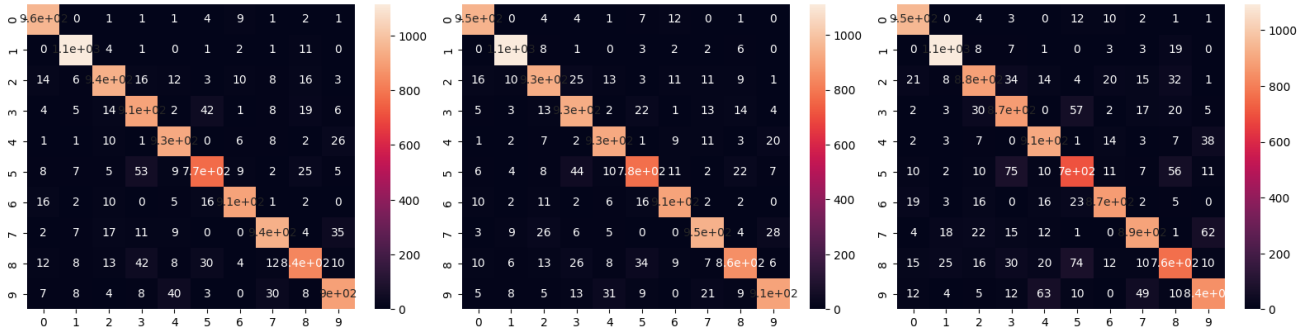
The test dataset was evaluated against all permutations of

1. **SVM:** Linear and Kernel
2. **Kernel:** RBF and Polynomial
3. **Dimensionality Reduction:** PCA and LDA

The results can be summarized in the following table:

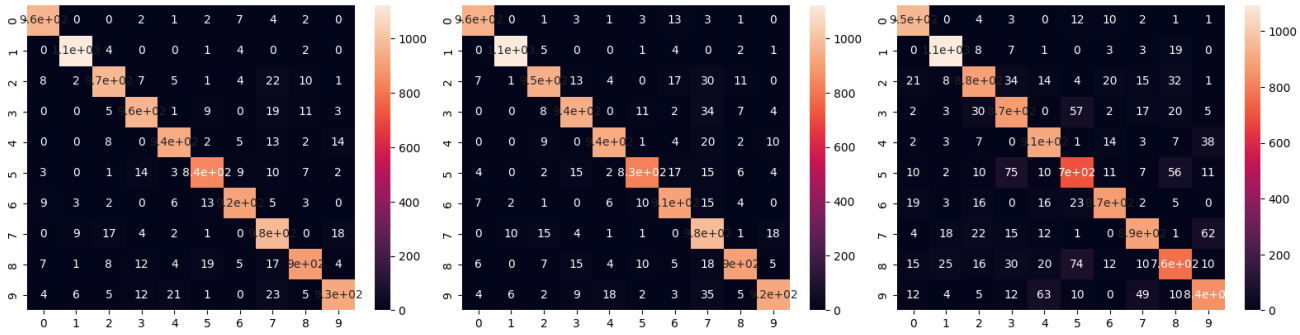
Table 1.4: SVM all combinations results

SVM									
Type	**Linear**			**Kernel**					
				RBF			**Polynomial**		
Dim.Reduction	**None**	**PCA**	**LDA**	None	**PCA**	**LDA**	**None**	**PCA**	**LDA**
Train Accuraccy	99.94	96.33	89.74	98.52	96.54	89.74	96.21	11.62	89.74
Test Accuracy	92.09	93	87.78	**95.24**	95.24	87.78	93.22	12	87.78



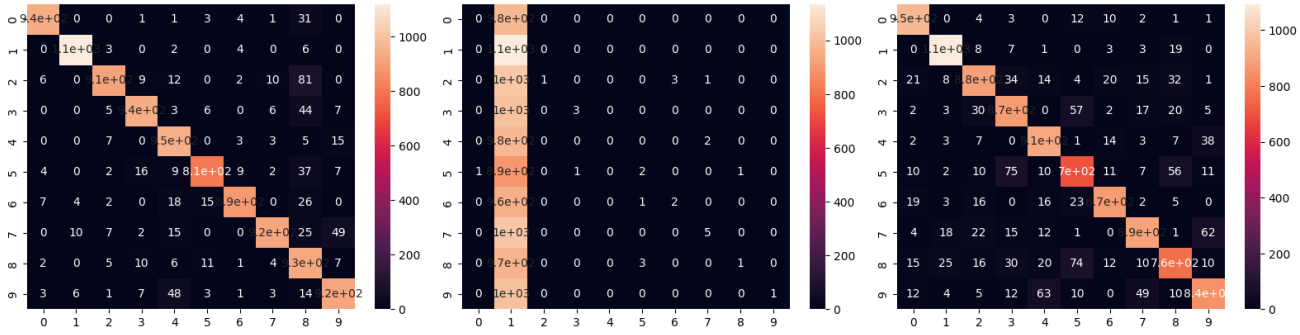
(a) No Dimensionality Reduction **(b) With PCA** **(c) With LDA**

Figure 1.1: Linear SVM Confusion Matrices



(a) No Dimensionality Reduction **(b) With PCA** **(c) With LDA**

Figure 1.2: Kernel SVM with RBF Kernel Confusion Matrices



(a) No Dimensionality Reduction (b) With PCA (c) With LDA

Figure 1.3: Kernel SVM with Polynomial Kernel Confusion Matrices

1.2 LOGISTIC REGRESSION

Implements multiclass logistic regression for MNIST. Multiclass logistic regression models the class posteriors of an image log linearly in terms of its features so that

$$P(y_{m,n} = 1 \mid \mathbf{x}_n) = \frac{\exp\left(\theta_m^\top \mathbf{x}_n\right)}{\sum_{i=1}^{M} \exp\left(\theta_i^\top \mathbf{x}_n\right)}$$

i.e. the softmax function, where $y_{m,n}$ is the indicator of x_n being in class ω_m. The model parameters are estimated to minimize the negative log-likelihood of the class labels, which are encoded as categorical vectors. The negative log-likelihood takes the form

$$\ell(\theta_1, \cdots, \theta_M) = -\sum_{n=1}^{N} \sum_{m=1}^{M} y_{m,n} \ln \left(\frac{\exp\left(\theta_m^\top \mathbf{x}_n\right)}{\sum_{i=1}^{M} \exp\left(\theta_i^\top \mathbf{x}_n\right)} \right)$$

Implementation :

For logistic regression, the data standardization technique applied was the same as the one used in SVM. Also, the data was sampled similarly with 20,000 samples for training and 10,000 samples for testing. Here, when training the model, it was passed in a batch size of 500 samples for every iteration. The output shown is after training in 15 epochs/iterations.

One interesting observation noticed in the training of logistic regressor for MNIST was that the implemented negative log-likelihood loss explodes after 20 iterations, this was because the gradients explode after 20 iterations. Thus the iterations were restricted to 15 epochs which gave an accuracy above 90%

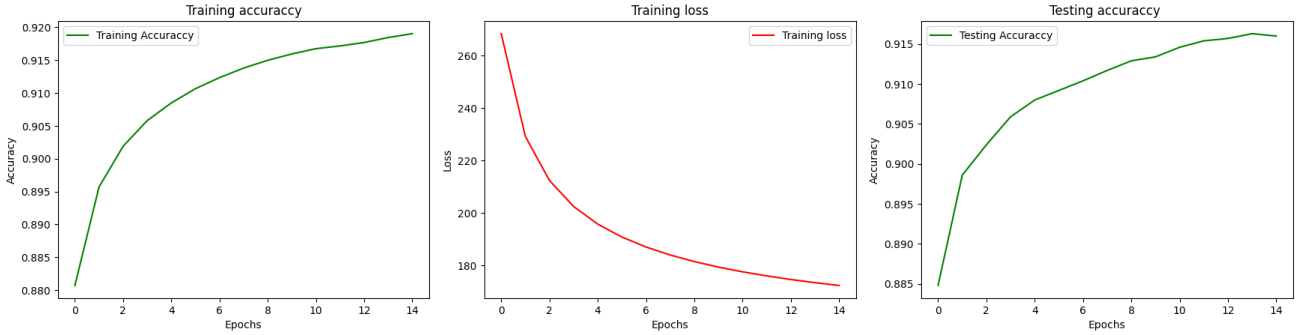
The hyperparameters used to get the following best results are:

1. **Learning Rate / Step size:** 0.0025
2. **Batch size:** 500
3. **Epoch:** 15
3. **Sampled dataset size:** 60,000 training samples and 10,000 testing samples

The results can be seen in the table 1.5

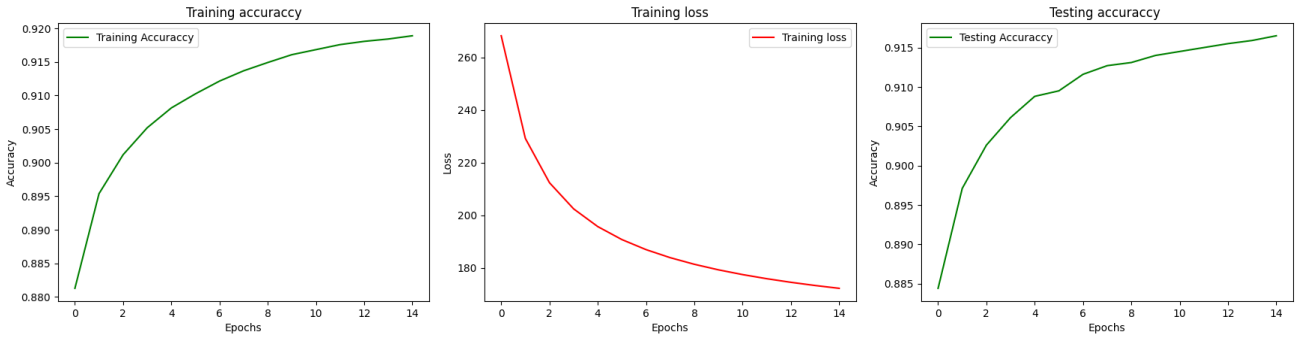
Table 1.5: Logistic Regression Results

Logistic Regression			
Dimensionality Reduction	None	PCA (rbf kernel)	LDA (rbf kernel)
Training Accuracy	92.37	91.88	88.78
Testing Accuracy	**91.68**	91.65	86.87



(a) Training Accuracy **(b) Training Loss** **(c) Test Accuracy**

Figure 1.4: Logistic Regression No dim. reduction



(a) Training Accuracy **(b) Training Loss** **(c) Test Accuracy**

Figure 1.5: Logistic Regression with PCA

(a) Training Accuracy **(b) Training Loss** **(c) Test Accuracy**

Figure 1.6: Logistic Regression with LDA

1.3 DEEP LEARNING:

Here, we build a Convolutional Neural Network using Pytorch and train it on the MNIST dataset.

Implementation

Here, a neural network is implemented with seven convolutional layers. The neural net is a sequence of a repeating block consisting of a convolutional layer, Batch normalization layer, ReLu and occasional MaxPool. The reason to choose the primitive block architecture was to build a very simplified version of the Deep ResNet model which has a more complicated structure and is much deeper. To help the model generalize well, a Dropout layer is added in the final block. The network architecture can be seen in table 1.6

Observations:

CNN outputs show a significant increase in accuracy compared to SVM and Logistic regression. We can conclude that, in order to classify image datasets, CNN proves to be a better choice.

Hyperparameters: The model was trained on two configurations as follows:

- Stochastic Gradient Descent (SGD):
  1. Learning Rate = 0.0025
  2. Momentum = 0.9
  3. Weight_Decay (L2 penalty) = None
  4. Nesterov = True

  Validation Accuracy = 99.34

  Test Accuracy = 99.17

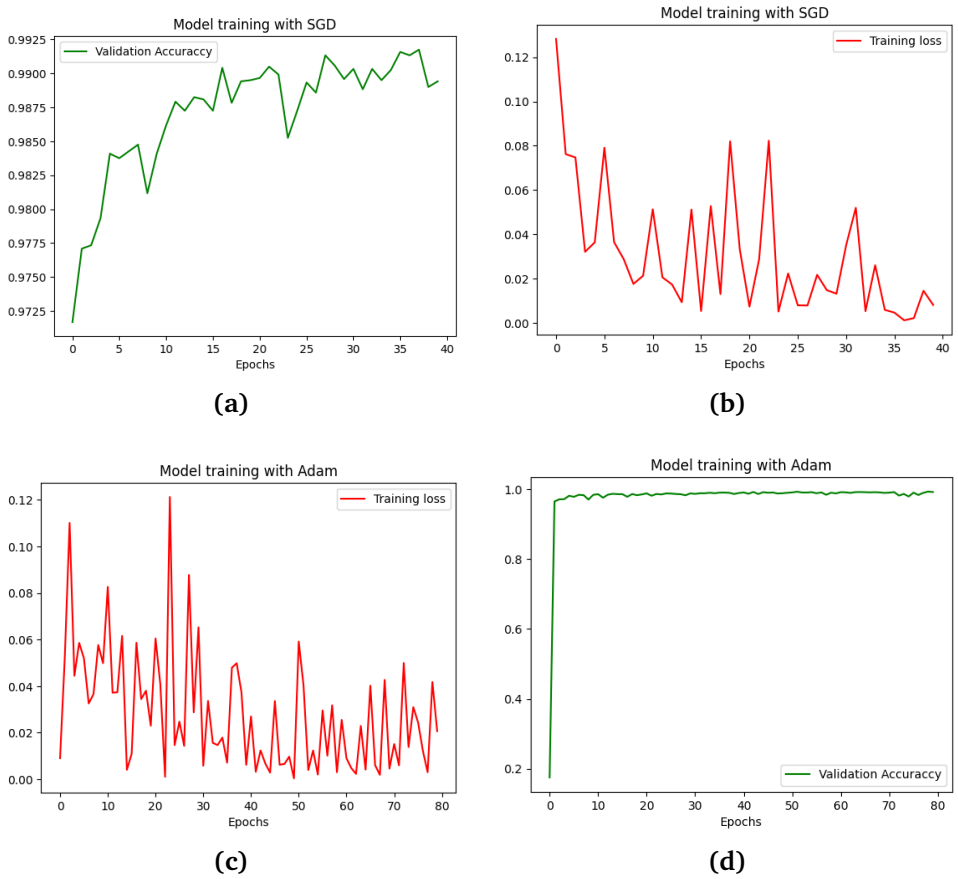


Figure 1.7: CNN for MNIST accuracy and loss.

- Adam:
  1. Learning Rate = 0.0015
  2. Betas = (0.9, 0.999)
  3. eps (for numeric stability) = 1e-8
  4. Weight_Decay (L2 penalty) = None

  Validation Accuracy = 99.27

  Test Accuracy = 99.32

We can see that with the Adam optimizer model we have observed the best test accuracy for MNIST so far which is equal to **99.32%**

Table 1.6: CNN architecture for MNIST classification

Sr	Layer
(0)	**Conv2d**(1, 32, kernel_size=(3, 3), stride=(1, 1))
(1)	**BatchNorm2d**(32, eps=1e-05, momentum=0.1, affine=True)
(2)	**ReLU**()
(3)	**Conv2d**(32, 64, kernel_size=(3, 3), stride=(1, 1))
(4)	**BatchNorm2d**(64, eps=1e-05, momentum=0.1, affine=True)
(5)	**ReLU**()
(6)	**MaxPool2d**(kernel_size=(2, 2), stride=(2, 2), padding=0)
(7)	**Conv2d**(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8)	**BatchNorm2d**(64, eps=1e-05, momentum=0.1, affine=True)
(9)	**ReLU**()
(10)	**Conv2d**(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11)	**BatchNorm2d**(64, eps=1e-05, momentum=0.1, affine=True)
(12)	**ReLU**()
(13)	**MaxPool2d**(kernel_size=(2, 2), stride=(2, 2), padding=0)
(14)	**Conv2d**(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15)	**BatchNorm2d**(128, eps=1e-05, momentum=0.1, affine=True)
(16)	**ReLU**()
(17)	**Conv2d**(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18)	**BatchNorm2d**(128, eps=1e-05, momentum=0.1, affine=True)
(19)	**ReLU**()
(20)	**Conv2d**(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(21)	**BatchNorm2d**(128, eps=1e-05, momentum=0.1, affine=True)
(22)	**ReLU**()
(23)	**MaxPool2d**(kernel_size=(2, 2), stride=(2, 2), padding=0)
(24)	**Flatten**()
(25)	**Dropout**(p=0.2, inplace=True)
(26)	**Linear**(in_features=1152, out_features=128, bias=True)
(27)	**ReLU**()
(28)	**Linear**(in_features=128, out_features=128, bias=True)
(29)	**ReLU**()
(30)	**Linear**(in_features=128, out_features=10, bias=True)

Chapter 2

Part II: Transfer Learning

In this part, we classify the images to one of the 10 species of monkeys using the 10 Monkey Species dataset. In section one, we train a simple CNN for classification and later compare it with results on the ResNet50 model fine-tuned on the underlying dataset.

2.1 SIMPLE CONVOLUTIONAL NEURAL NETWORK

Here a simple CNN is built using Pytorch and trained on CUDA. The dataset originally contains 1097 train and test images. 0.2% of the training dataset is split as the validation set. The total number of monkey classes are 10. All images are normalized with the known mean and standard deviation values of the dataset

Observations:

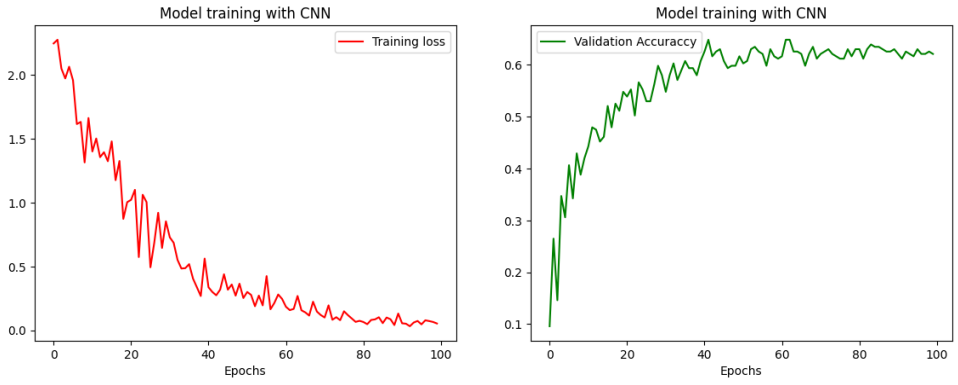
We saw that CNN showed good results to classify image dataset. But in this case, we do not have enough training data for the model to learn the classes. As a result, an **accuracy of 62.87%** was observed. The hyperparameters of the model are as follows

Hyperparameters: The model was trained on two configurations as follows:

- Stochastic Gradient Descent (SGD):
  1. Learning Rate = 0.00215
  2. Momentum = 0.9
  3. Weight_Decay (L2 penalty) = None
  4. Nesterov = True

  Validation Accuracy = 64.84%

  Test Accuracy = 62.87%



(a) (b)

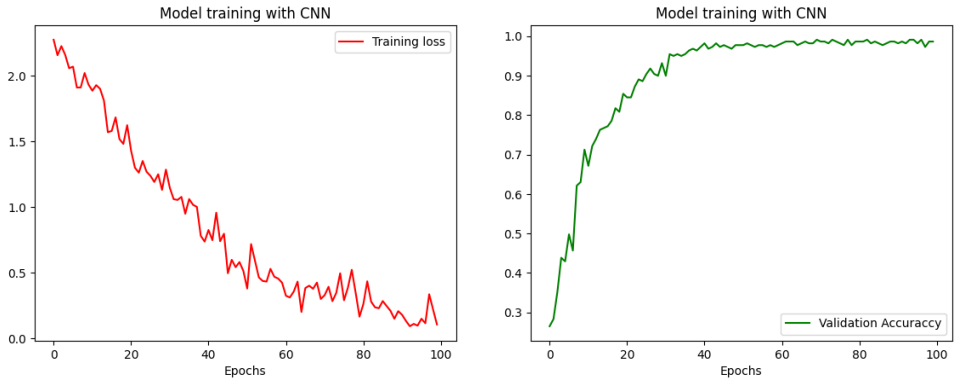
Figure 2.1: CNN for KITTI Monkey accuracy and loss.

Table 2.1: CNN architecture for Monkey dataset

Sr	Layer
(0)	**Conv2d**(3, 32, kernel_size=(3, 3), stride=(1, 1))
(1)	**ReLU**()
(2)	**ReLU**(kernel_size=(2, 2), stride=(2, 2), padding=0)
(3)	**Conv2d**(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(4)	**BatchNorm2d**(64, eps=1e-05, momentum=0.1, affine=True)
(5)	**ReLU**()
(6)	**Conv2d**(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7)	**BatchNorm2d**(128, eps=1e-05, momentum=0.1, affine=True)
(8)	**ReLU**()
(9)	**ReLU**(kernel_size=(2, 2), stride=(2, 2), padding=0)
(10)	**Flatten**()
(11)	**Linear**(in_features=4608, out_features=32, bias=True)
(12)	**ReLU**()
(13)	**Linear**(in_features=32, out_features=10, bias=True)

2.2 TRANSFER LEARNING USING RESNET50

The low accuracy that we got with a simple CNN can be increased with a deeper network that has been trained on a wider dataset of images. ResNet50 is a model with 50 layers, trained to 1000 different classes. Here we load this model and fine-tune it by training just the final layer with the monkey dataset. For fine-tuning, we freeze the model parameters and then replace the final FC layer with an untrained



(a) **(b)**

Figure 2.2: KITTI Monkey dataset transfer learning plots.

one. After this, the model is trained on the monkey dataset. This was also trained on CUDA and with the **same hyperparameters** as the simple CNN to compare the results.

Observations: The **increase in accuracy was 36.39%**. The model showed almost perfect training and testing accuracy.

Training accuracy = 99.08

Testing Accuracy = 99.26

The plot of training loss and validation accuracy can be seen in figure 2.2. It can also be observed that the accuracy cure is relatively smoother here compared to the simple CNN results.