Deep Learning Assignment 1

Name: Dinesh Adhithya

Registration No./Roll No.: 18097

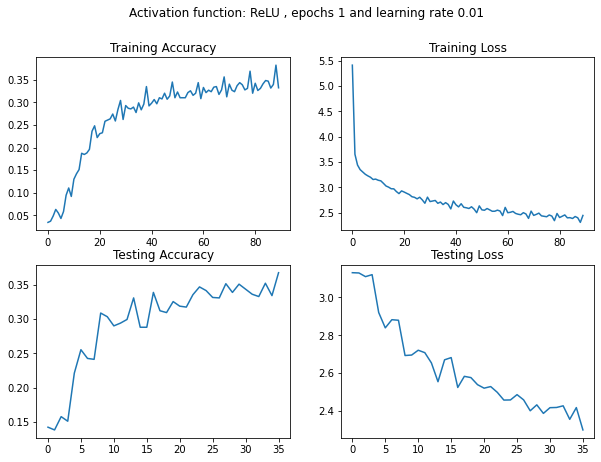
Institute/University Name: IISER Bhopal

Program/Stream: EECS

Question 1

A Feed forward Neural Network model was designed , Xavier initialization was done for the weights and cross entropy as loss function and ADAM as optimizer

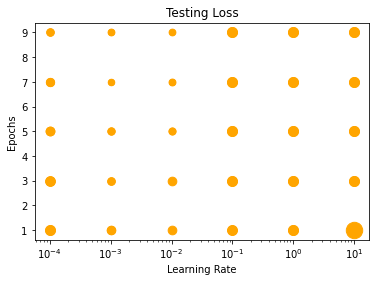
The Model's performance for hyper parameters : ReLU activation function , epoch 1 and learning rate 0.01.

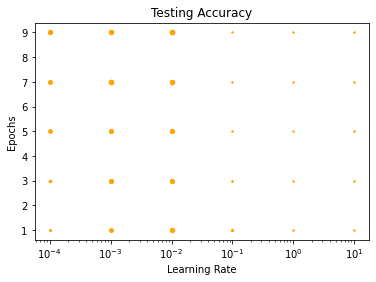


For finding the best performing Model we wish to tune our hyperparameters and find the best combination of hyperparameters based on testing accuracy:

Now the model was trained for various hyperparameters and based on testing loss and accuracy hyperparameters shall be chosen.

For below figures larger circle size means larger value.

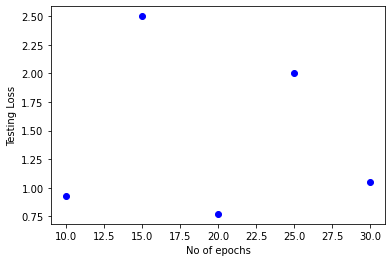




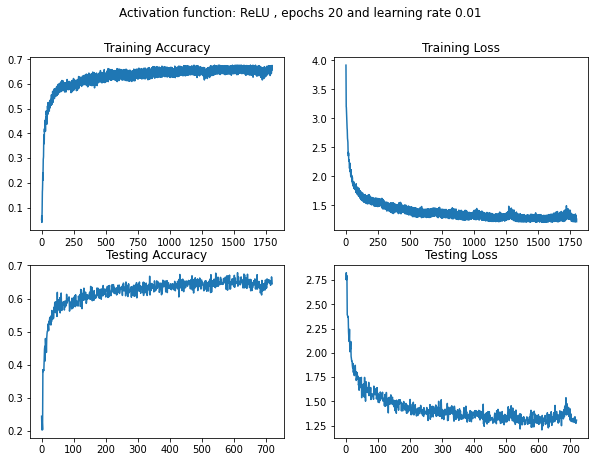
Now we understand RELU and 0.01 learning rate are the best choice. Now changing epochs from 10,15,20,25 and 30.







Best performing combination of hyperparameters was: 20 epochs , 0.01 learning rate and ReLU activation function.



Now let's try to reason the observation:

1. activation function : ReLU performs better as it can converge faster than Tanh.
2. Learning rate is 0.01 , large learning rate can lead to large loss and gradient explosion , small learning rate will take too long to converge, 0.01 is quite the perfect learning rate.
3. Epochs happen to be 20 , if the epoch is too large the model can overfit and if it happens to be too less model won't be able to learn the features quite well.

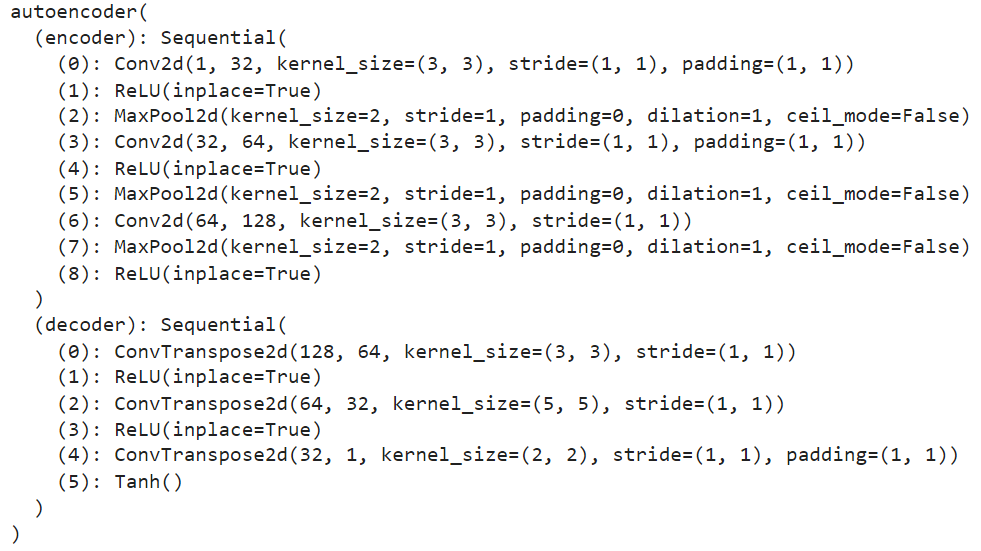
Question 2

I had designed a CNN based autoencoder , CNN layers are used to extract low level features and later a deconvolution operation is done to upsample the image to its original size. With 3 encoding and 3 decoding layers and MSE as loss function.

Noise is added to Images and passed as input:



Model Architecture:



Model Performance:

EPOCHS: 1 LOSS VALUE: 0.006791803054511547

EPOCHS: 1 LOSS VALUE TEST DATA: 0.006614283192902803

EPOCHS: 1 LOSS VALUE TEST DATA: 0.006600106135010719

EPOCHS: 1 LOSS VALUE TEST DATA: 0.006580515764653683

EPOCHS: 1 LOSS VALUE TEST DATA: 0.006512995343655348

EPOCHS: 1 LOSS VALUE TEST DATA: 0.006616662722080946

EPOCHS: 2 LOSS VALUE: 0.005858926568180323

EPOCHS: 2 LOSS VALUE TEST DATA: 0.005708599463105202

EPOCHS: 2 LOSS VALUE TEST DATA: 0.005709699355065823

EPOCHS: 2 LOSS VALUE TEST DATA: 0.00572435325011611

EPOCHS: 2 LOSS VALUE TEST DATA: 0.00574338436126709

EPOCHS: 2 LOSS VALUE TEST DATA: 0.005736910738050938

EPOCHS: 3 LOSS VALUE: 0.0054824938997626305

EPOCHS: 3 LOSS VALUE TEST DATA: 0.005348770879209042

EPOCHS: 3 LOSS VALUE TEST DATA: 0.005353259854018688

EPOCHS: 3 LOSS VALUE TEST DATA: 0.005370001308619976

EPOCHS: 3 LOSS VALUE TEST DATA: 0.0053702606819570065

EPOCHS: 3 LOSS VALUE TEST DATA: 0.005351599771529436

EPOCHS: 4 LOSS VALUE: 0.005212779156863689

EPOCHS: 4 LOSS VALUE TEST DATA: 0.005143256857991219

EPOCHS: 4 LOSS VALUE TEST DATA: 0.005122030153870583

EPOCHS: 4 LOSS VALUE TEST DATA: 0.005142623092979193

EPOCHS: 4 LOSS VALUE TEST DATA: 0.00513771828263998

EPOCHS: 4 LOSS VALUE TEST DATA: 0.005136240739375353

EPOCHS: 5 LOSS VALUE: 0.005052964668720961

EPOCHS: 5 LOSS VALUE TEST DATA: 0.004988811910152435

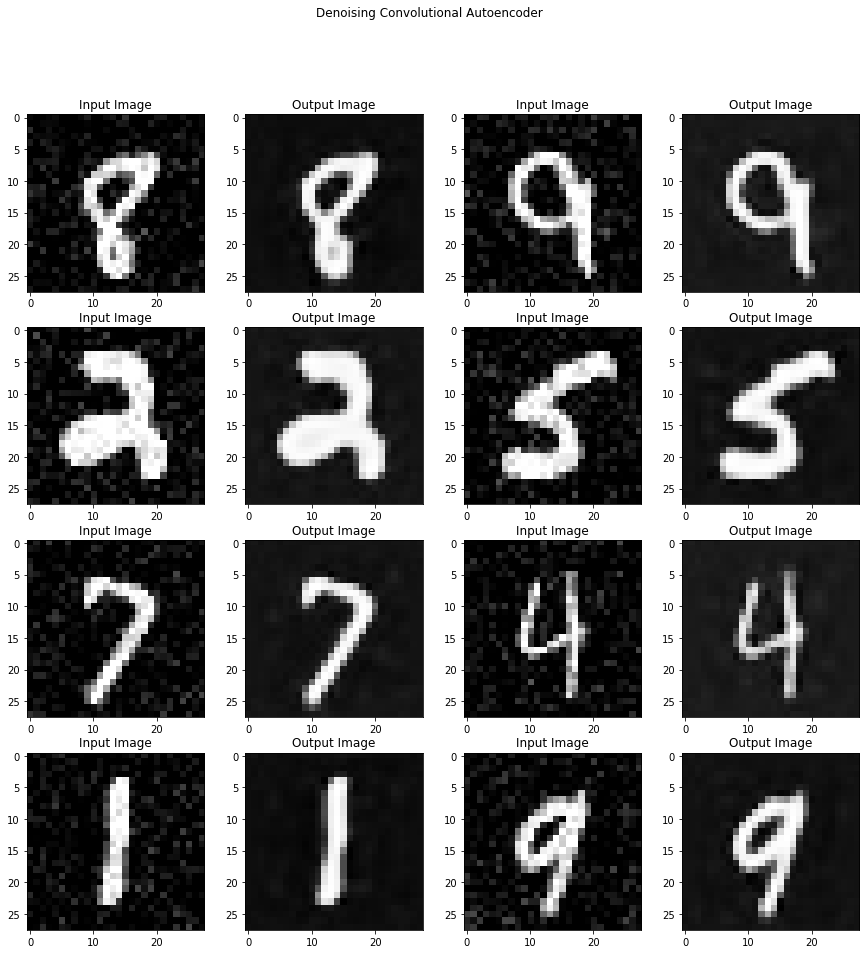
EPOCHS: 5 LOSS VALUE TEST DATA: 0.004973393864929676

EPOCHS: 5 LOSS VALUE TEST DATA: 0.004985738079994917

EPOCHS: 5 LOSS VALUE TEST DATA: 0.004979907535016537

EPOCHS: 5 LOSS VALUE TEST DATA: 0.004947187379002571

Output Images vs Noisy Input Image



### A. Use 1 FC layer with softmax activation for 10-class classification

Model Architecture:

The output of the encoder is passed to a single fully connected layer. No need to have softmax as activation function as the optimizer nn.CrossEntropyloss already implements it.

class NeuralNetwork1(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork1,self).\_\_init\_\_()

self.fc1=nn.Linear(67712,10)

def forward(self,x):

latent\_vector=model.encoder(x).view(len(x),-1)

o=self.fc1(latent\_vector)

return o

Model Performance:

EPOCHS: 1 Iteration: 1 LOSS VALUE TRAIN DATA: 0.9098833799362183 ACCURACY TRAIN DATA: 0.8622

EPOCHS: 1 Iteration: 1 LOSS VALUE TEST DATA: 0.8815616965293884 ACCURACY TEST DATA: 0.8362

EPOCHS: 1 Iteration: 2 LOSS VALUE TRAIN DATA: 0.45752426981925964 ACCURACY TRAIN DATA: 0.8896

EPOCHS: 1 Iteration: 2 LOSS VALUE TEST DATA: 0.507055401802063 ACCURACY TEST DATA: 0.8676

EPOCHS: 1 Iteration: 3 LOSS VALUE TRAIN DATA: 0.36103856563568115 ACCURACY TRAIN DATA: 0.908

EPOCHS: 1 Iteration: 3 LOSS VALUE TEST DATA: 0.43281909823417664 ACCURACY TEST DATA: 0.881

EPOCHS: 1 Iteration: 4 LOSS VALUE TRAIN DATA: 0.32594621181488037 ACCURACY TRAIN DATA: 0.916

EPOCHS: 1 Iteration: 4 LOSS VALUE TEST DATA: 0.39402738213539124 ACCURACY TEST DATA: 0.8902

EPOCHS: 1 Iteration: 5 LOSS VALUE TRAIN DATA: 0.30496400594711304 ACCURACY TRAIN DATA: 0.919

EPOCHS: 1 Iteration: 5 LOSS VALUE TEST DATA: 0.36595427989959717 ACCURACY TEST DATA: 0.8998

EPOCHS: 1 Iteration: 6 LOSS VALUE TRAIN DATA: 0.2904203534126282 ACCURACY TRAIN DATA: 0.9238

EPOCHS: 1 Iteration: 6 LOSS VALUE TEST DATA: 0.34920838475227356 ACCURACY TEST DATA: 0.9038

EPOCHS: 1 Iteration: 7 LOSS VALUE TRAIN DATA: 0.27683743834495544 ACCURACY TRAIN DATA: 0.927

EPOCHS: 1 Iteration: 7 LOSS VALUE TEST DATA: 0.33869174122810364 ACCURACY TEST DATA: 0.907

EPOCHS: 1 Iteration: 8 LOSS VALUE TRAIN DATA: 0.2716909945011139 ACCURACY TRAIN DATA: 0.9294

EPOCHS: 1 Iteration: 8 LOSS VALUE TEST DATA: 0.33195745944976807 ACCURACY TEST DATA: 0.9112

EPOCHS: 1 Iteration: 9 LOSS VALUE TRAIN DATA: 0.2622767686843872 ACCURACY TRAIN DATA: 0.9306

EPOCHS: 1 Iteration: 9 LOSS VALUE TEST DATA: 0.30681371688842773 ACCURACY TEST DATA: 0.9192

EPOCHS: 1 Iteration: 10 LOSS VALUE TRAIN DATA: 0.255996435880661 ACCURACY TRAIN DATA: 0.9342

EPOCHS: 1 Iteration: 10 LOSS VALUE TEST DATA: 0.31170007586479187 ACCURACY TEST DATA: 0.914

EPOCHS: 1 Iteration: 11 LOSS VALUE TRAIN DATA: 0.24943411350250244 ACCURACY TRAIN DATA: 0.9342

EPOCHS: 1 Iteration: 11 LOSS VALUE TEST DATA: 0.2993046045303345 ACCURACY TEST DATA: 0.9206

EPOCHS: 1 Iteration: 12 LOSS VALUE TRAIN DATA: 0.24525542557239532 ACCURACY TRAIN DATA: 0.9352

EPOCHS: 1 Iteration: 12 LOSS VALUE TEST DATA: 0.2943926751613617 ACCURACY TEST DATA: 0.9216

### B. Use 2 FC layers with softmax activation for 10-class classification

The output of the encoder is passed to a fully connected layer and then another fully connected layer . No need to have softmax as activation function as the optimizer nn.CrossEntropyloss already implements it.

Model Architecture:

class NeuralNetwork2(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork2,self).\_\_init\_\_()

self.fc1=nn.Linear(67712,50)

self.fc2=nn.Linear(50,10)

def forward(self,x):

latent\_vector=model.encoder(x).view(len(x),-1)

o=self.fc1(latent\_vector)

o=self.fc2(o)

return o

Model Performance:

EPOCHS: 1 Iteration: 1 LOSS VALUE TRAIN DATA: 1.6122562885284424 ACCURACY TRAIN DATA: 0.712

EPOCHS: 1 Iteration: 1 LOSS VALUE TEST DATA: 1.5563935041427612 ACCURACY TEST DATA: 0.6944

EPOCHS: 1 Iteration: 2 LOSS VALUE TRAIN DATA: 0.8589674830436707 ACCURACY TRAIN DATA: 0.8322

EPOCHS: 1 Iteration: 2 LOSS VALUE TEST DATA: 0.8828794956207275 ACCURACY TEST DATA: 0.7998

EPOCHS: 1 Iteration: 3 LOSS VALUE TRAIN DATA: 0.5368932485580444 ACCURACY TRAIN DATA: 0.8818

EPOCHS: 1 Iteration: 3 LOSS VALUE TEST DATA: 0.6004245281219482 ACCURACY TEST DATA: 0.848

EPOCHS: 1 Iteration: 4 LOSS VALUE TRAIN DATA: 0.41248834133148193 ACCURACY TRAIN DATA: 0.8988

EPOCHS: 1 Iteration: 4 LOSS VALUE TEST DATA: 0.4739094376564026 ACCURACY TEST DATA: 0.8688

EPOCHS: 1 Iteration: 5 LOSS VALUE TRAIN DATA: 0.35387274622917175 ACCURACY TRAIN DATA: 0.9102

EPOCHS: 1 Iteration: 5 LOSS VALUE TEST DATA: 0.414013534784317 ACCURACY TEST DATA: 0.89

EPOCHS: 1 Iteration: 6 LOSS VALUE TRAIN DATA: 0.32246899604797363 ACCURACY TRAIN DATA: 0.9144

EPOCHS: 1 Iteration: 6 LOSS VALUE TEST DATA: 0.3871513307094574 ACCURACY TEST DATA: 0.8938

EPOCHS: 1 Iteration: 7 LOSS VALUE TRAIN DATA: 0.29997265338897705 ACCURACY TRAIN DATA: 0.9198

EPOCHS: 1 Iteration: 7 LOSS VALUE TEST DATA: 0.3555159866809845 ACCURACY TEST DATA: 0.901

EPOCHS: 1 Iteration: 8 LOSS VALUE TRAIN DATA: 0.2840052843093872 ACCURACY TRAIN DATA: 0.925

EPOCHS: 1 Iteration: 8 LOSS VALUE TEST DATA: 0.3435658812522888 ACCURACY TEST DATA: 0.9046

EPOCHS: 1 Iteration: 9 LOSS VALUE TRAIN DATA: 0.270526647567749 ACCURACY TRAIN DATA: 0.9256

EPOCHS: 1 Iteration: 9 LOSS VALUE TEST DATA: 0.3346247673034668 ACCURACY TEST DATA: 0.9048

EPOCHS: 1 Iteration: 10 LOSS VALUE TRAIN DATA: 0.2611357867717743 ACCURACY TRAIN DATA: 0.9288

EPOCHS: 1 Iteration: 10 LOSS VALUE TEST DATA: 0.31963425874710083 ACCURACY TEST DATA: 0.912

EPOCHS: 1 Iteration: 11 LOSS VALUE TRAIN DATA: 0.25201788544654846 ACCURACY TRAIN DATA: 0.9332

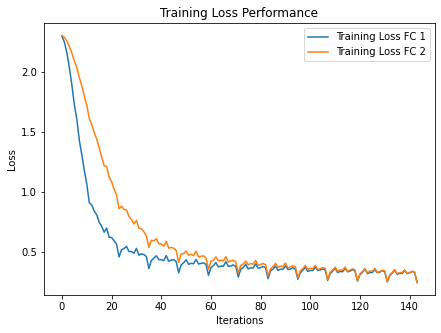
EPOCHS: 1 Iteration: 11 LOSS VALUE TEST DATA: 0.30735349655151367 ACCURACY TEST DATA: 0.9134

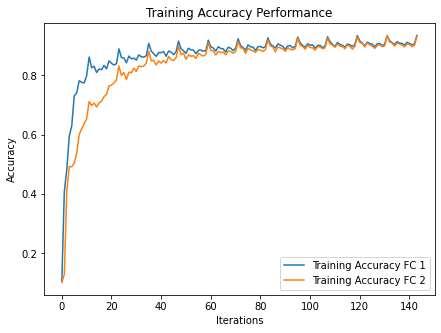
EPOCHS: 1 Iteration: 12 LOSS VALUE TRAIN DATA: 0.24502763152122498 ACCURACY TRAIN DATA: 0.9318

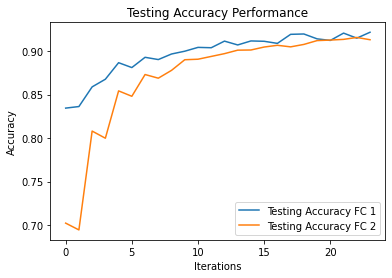
EPOCHS: 1 Iteration: 12 LOSS VALUE TEST DATA: 0.3044353127479553 ACCURACY TEST DATA: 0.913

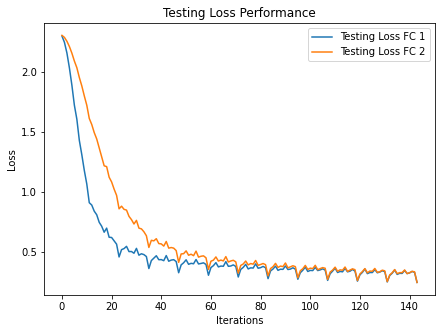
C. Compare the performance between 1FC and 2 FC layer results and

report the accuracy on test set and plot loss curves on training and test dataset









We can clearly see that the autoencoder model learns really well from the dataset as the outputs are very accurate hence it learns the latent space representation of images of various classes really well. Hence it is able to understand the low level features in the image and map them well to high level feature ie. the number in the image.

Using a single fully connected layer helps in quick convergence and better testing performance as the autoencoder itself has learned really well , whereas the 2 fully connected architecture takes some time to learn as it has unnecessarily more model parameters.

Question 3

A. Conv-Pool-Conv-Pool-Conv-Pool

Model Architecture:

class A(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.conv1=nn.Conv2d(3,6,5)

self.pool1=nn.MaxPool2d(2,2)

self.conv2=nn.Conv2d(6,14,5)

self.conv3=nn.Conv2d(14,10,3)

def forward(self,x):

x=self.pool1(self.conv1(x))

x=self.pool1(self.conv2(x))

x=self.pool1(self.conv3(x))

x = torch.flatten(x, 1)

return x

Training Performance:

[1, 2000] loss: 1.985

[1, 4000] loss: 1.763

[1, 6000] loss: 1.662

[1, 8000] loss: 1.616

[1, 10000] loss: 1.575

[2, 2000] loss: 1.518

[2, 4000] loss: 1.531

[2, 6000] loss: 1.503

[2, 8000] loss: 1.471

[2, 10000] loss: 1.464

[3, 2000] loss: 1.452

[3, 4000] loss: 1.442

[3, 6000] loss: 1.431

[3, 8000] loss: 1.422

[3, 10000] loss: 1.412

[4, 2000] loss: 1.393

[4, 4000] loss: 1.391

[4, 6000] loss: 1.396

[4, 8000] loss: 1.398

[4, 10000] loss: 1.398

[5, 2000] loss: 1.379

[5, 4000] loss: 1.378

[5, 6000] loss: 1.387

[5, 8000] loss: 1.355

[5, 10000] loss: 1.386

[6, 2000] loss: 1.328

[6, 4000] loss: 1.365

[6, 6000] loss: 1.353

[6, 8000] loss: 1.383

[6, 10000] loss: 1.370

[7, 2000] loss: 1.322

[7, 4000] loss: 1.345

[7, 6000] loss: 1.367

[7, 8000] loss: 1.326

[7, 10000] loss: 1.371

[8, 2000] loss: 1.324

[8, 4000] loss: 1.345

[8, 6000] loss: 1.341

[8, 8000] loss: 1.345

[8, 10000] loss: 1.322

[9, 2000] loss: 1.312

[9, 4000] loss: 1.342

[9, 6000] loss: 1.344

[9, 8000] loss: 1.330

[9, 10000] loss: 1.329

[10, 2000] loss: 1.308

[10, 4000] loss: 1.347

[10, 6000] loss: 1.329

[10, 8000] loss: 1.326

[10, 10000] loss: 1.327

Test Performance:

Accuracy of class plane : 0.4988009592326139

Accuracy of class car : 0.6062176165803109

Accuracy of class bird : 0.4358353510895884

Accuracy of class cat : 0.25984251968503935

Accuracy of class deer : 0.3263157894736842

Accuracy of class dog : 0.4010152284263959

Accuracy of class frog : 0.7493540051679587

Accuracy of class horse : 0.6310679611650486

Accuracy of class ship : 0.5980392156862745

Accuracy of class truck : 0.6398104265402843

Macro Average accuracy: 0.51675

B. Conv-Conv-Pool-Conv-Conv-Pool

Model Architecture:

class B(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.conv1=nn.Conv2d(3,6,5)

self.pool1=nn.MaxPool2d(2,2)

self.conv2=nn.Conv2d(6,10,5)

self.conv3=nn.Conv2d(10,3,2)

self.conv4=nn.Conv2d(3,10,10)

def forward(self,x):

x=self.conv1(x)

x=self.pool1(self.conv2(x))

x=self.conv3(x)

x=self.pool1(self.conv4(x))

x = torch.flatten(x, 1)

return x

Training Performance:

[1, 2000] loss: 1.942

[1, 4000] loss: 1.765

[1, 6000] loss: 1.684

[1, 8000] loss: 1.609

[1, 10000] loss: 1.569

[2, 2000] loss: 1.566

[2, 4000] loss: 1.550

[2, 6000] loss: 1.531

[2, 8000] loss: 1.513

[2, 10000] loss: 1.506

[3, 2000] loss: 1.473

[3, 4000] loss: 1.492

[3, 6000] loss: 1.478

[3, 8000] loss: 1.490

[3, 10000] loss: 1.473

[4, 2000] loss: 1.440

[4, 4000] loss: 1.455

[4, 6000] loss: 1.479

[4, 8000] loss: 1.462

[4, 10000] loss: 1.470

[5, 2000] loss: 1.438

[5, 4000] loss: 1.447

[5, 6000] loss: 1.443

[5, 8000] loss: 1.468

[5, 10000] loss: 1.446

[6, 2000] loss: 1.433

[6, 4000] loss: 1.440

[6, 6000] loss: 1.443

[6, 8000] loss: 1.444

[6, 10000] loss: 1.441

[7, 2000] loss: 1.424

[7, 4000] loss: 1.440

[7, 6000] loss: 1.431

[7, 8000] loss: 1.439

[7, 10000] loss: 1.449

[8, 2000] loss: 1.440

[8, 4000] loss: 1.441

[8, 6000] loss: 1.424

[8, 8000] loss: 1.412

[8, 10000] loss: 1.425

[9, 2000] loss: 1.424

[9, 4000] loss: 1.427

[9, 6000] loss: 1.429

[9, 8000] loss: 1.429

[9, 10000] loss: 1.438

[10, 2000] loss: 1.414

[10, 4000] loss: 1.432

[10, 6000] loss: 1.403

[10, 8000] loss: 1.422

[10, 10000] loss: 1.432

Testing Performance:

Accuracy of class plane : 0.5971223021582733

Accuracy of class car : 0.5958549222797928

Accuracy of class bird : 0.43825665859564167

Accuracy of class cat : 0.33070866141732286

Accuracy of class deer : 0.3631578947368421

Accuracy of class dog : 0.3604060913705584

Accuracy of class frog : 0.6744186046511628

Accuracy of class horse : 0.49514563106796117

Accuracy of class ship : 0.5784313725490197

Accuracy of class truck : 0.6492890995260664

Macro Average accuracy: 0.51025

C. Conv-Pool-Conv-Pool-Conv-Pool-FC-FC

Model Performance:

class C(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.conv1=nn.Conv2d(3,6,5)

self.pool1=nn.MaxPool2d(2,2)

self.conv2=nn.Conv2d(6,16,5)

self.conv3=nn.Conv2d(16,20,2)

self.fc1=nn.Linear(80,40)

self.fc2=nn.Linear(40,10)

def forward(self,x):

x=self.pool1(self.conv1(x))

x=self.pool1(self.conv2(x))

x=self.pool1(self.conv3(x))

x = torch.flatten(x, 1)

x=self.fc1(x)

x=self.fc2(x)

return x

Training Performance:

[1, 2000] loss: 2.025

[1, 4000] loss: 1.760

[1, 6000] loss: 1.606

[1, 8000] loss: 1.549

[1, 10000] loss: 1.485

[2, 2000] loss: 1.470

[2, 4000] loss: 1.442

[2, 6000] loss: 1.419

[2, 8000] loss: 1.403

[2, 10000] loss: 1.381

[3, 2000] loss: 1.371

[3, 4000] loss: 1.364

[3, 6000] loss: 1.346

[3, 8000] loss: 1.334

[3, 10000] loss: 1.311

[4, 2000] loss: 1.287

[4, 4000] loss: 1.302

[4, 6000] loss: 1.290

[4, 8000] loss: 1.303

[4, 10000] loss: 1.283

[5, 2000] loss: 1.265

[5, 4000] loss: 1.260

[5, 6000] loss: 1.263

[5, 8000] loss: 1.256

[5, 10000] loss: 1.252

[6, 2000] loss: 1.218

[6, 4000] loss: 1.214

[6, 6000] loss: 1.239

[6, 8000] loss: 1.248

[6, 10000] loss: 1.238

[7, 2000] loss: 1.210

[7, 4000] loss: 1.216

[7, 6000] loss: 1.224

[7, 8000] loss: 1.215

[7, 10000] loss: 1.223

[8, 2000] loss: 1.175

[8, 4000] loss: 1.202

[8, 6000] loss: 1.211

[8, 8000] loss: 1.223

[8, 10000] loss: 1.215

[9, 2000] loss: 1.184

[9, 4000] loss: 1.178

[9, 6000] loss: 1.214

[9, 8000] loss: 1.181

[9, 10000] loss: 1.210

[10, 2000] loss: 1.177

[10, 4000] loss: 1.178

[10, 6000] loss: 1.185

[10, 8000] loss: 1.204

[10, 10000] loss: 1.196

Testing Performance:

Accuracy of class plane : 0.6354916067146283

Accuracy of class car : 0.7461139896373057

Accuracy of class bird : 0.3898305084745763

Accuracy of class cat : 0.29658792650918636

Accuracy of class deer : 0.45789473684210524

Accuracy of class dog : 0.5431472081218274

Accuracy of class frog : 0.8165374677002584

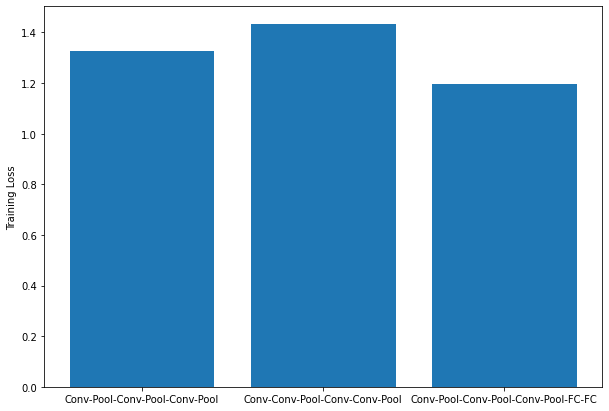
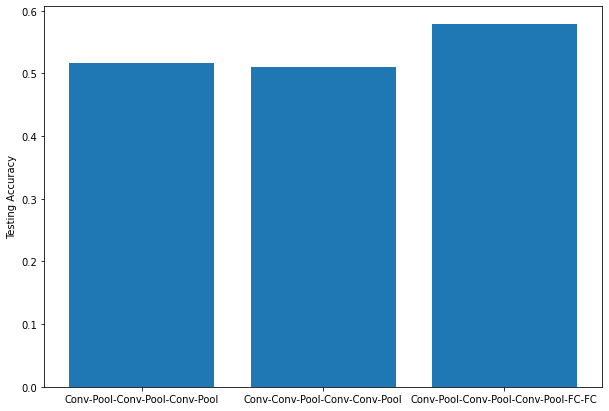
Accuracy of class horse : 0.6019417475728155

Accuracy of class ship : 0.7230392156862745

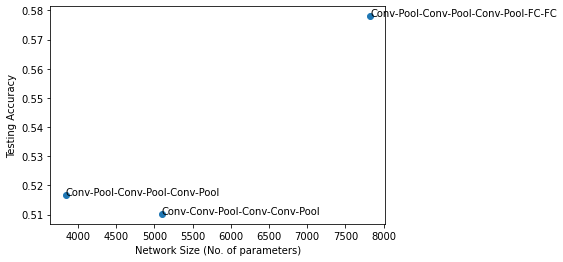
Accuracy of class truck : 0.5639810426540285

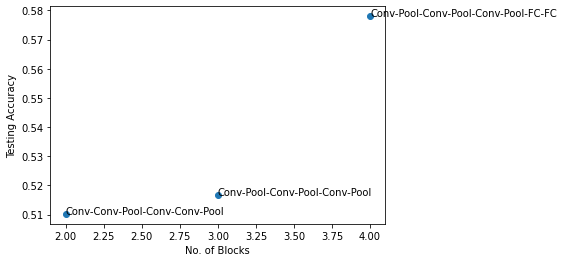
Macro Average accuracy: 0.578

Comparison of Model Performances:



A. How does changing the network size change the accuracy?





* Model C overperforms both the other models as it has fully connected layers which help in learning from low features extracted from previous CNN layers to high level features and helps in better classification.
* Model B in spite of having larger parameters than model A , performs worse than A. As model A has convolutional layers then subsampled down to smaller size , then again features are extracted by CNN and again pooled down to smaller size , this leads to systematic reduction of low level features to high level features and this helps in better learning of features.
* Whereas model B extracts features using CNN but is drastically down-sampled , because of only 2 pooling layers and this leads to larger loss of features compared to model A.
* We can clearly see that just because a model has larger parameters and larger blocks doesn't mean it would improve performance.

B. Experiment with different sizes of pooling and do a detailed analysis of

pooling size on the network.

We select the best performing model architecture ie. Conv-Pool-Conv-Pool-Conv-Pool-FC-FC , We change the pooling layer size from 2 to 8 and then analyze the testing accuracy and loss.

pooling\_size=[2,3,4,5,6,7,8]

out=[8000,5780,3920,2420,1280,500,80] , out is dependent on pooling size.

Model Architecture:

class D(nn.Module):

def \_\_init\_\_(self,pooling\_size1,out):

super().\_\_init\_\_()

self.conv1=nn.Conv2d(3,6,5)

self.pool1=nn.MaxPool2d(pooling\_size1,1)

self.conv2=nn.Conv2d(6,16,5)

self.conv3=nn.Conv2d(16,20,2)

self.fc1=nn.Linear(out,40)

self.fc2=nn.Linear(40,10)

def forward(self,x):

x=self.pool1(self.conv1(x))

x=self.pool1(self.conv2(x))

x=self.pool1(self.conv3(x))

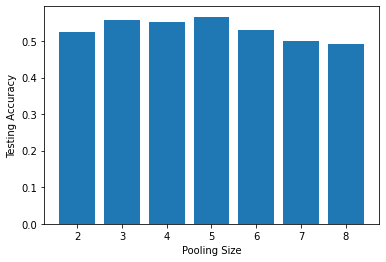
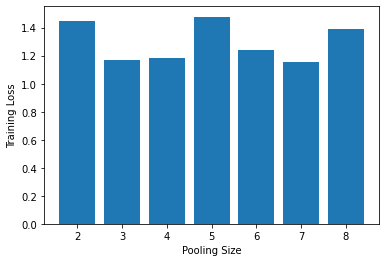
x= torch.flatten(x, 1)

x=self.fc1(x)

x=self.fc2(x)

return x

Performance of Model over various pooling sizes:



The pooling size over which both testing loss is minimum and testing accuracy is maximum is for pooling size 3 and 4 .

A small pooling size like 2 , may keep hold of more features which are redundant and results in slower learning , a large pooling size will lead to feature loss and the filters won’t be able to learn low level features quite well hence a pooling size of 3 and 4 brings good balance of faster learning and testing accuracy.

C. How the presence of one or more fully connected layers changes the

accuracy.

The addition of fully connected layers , helps in improving the testing accuracy from 0.51 to 0.57 , this is due to the fact that CNN’s are capable to extracting low level features using various filters and fully connected linear layers help in extracting high level features from the output of CNN’s , this helps in better classification of images.

Hence fully connected layers help in improving the model performance.

References:

1.<https://pytorch.org/>

2.<https://stackoverflow.com/>

3.Modern Computer Vision with PyTorch by V Kishore Ayyadevara, Yeshwanth Reddy