Objective: To create, train and test a Convolutional Neural Network based on LeNet-5 architecture for the task of Image classification using CIFAR-10 dataset.

1. Introduction

Artificial Neural Networks (ANN) popularly known as Multilayer Perceptrons were very efficient for raw data sets where the inputs were defined as vectors in d-dimensional space. Multilayer Perceptrons (MLPs) were very efficient in Pattern recognition and Classification tasks. But for images, it was a tedious task to achieve convergence on such a huge dimension image data-samples. Initially, the idea was to extract features (image descriptors) from images and define an image using a single feature vector. Scale Invariant Feature Transform (SIFT), Histogram of Gradients (HoG) were popular feature extraction algorithms which were used widely in all High-Level vision tasks. However, the idea of extracting features from images automatically without the intervention of humans (hand-crafted features) led to the development of an end-to-end architecture. LeCun et.al proposed a state-of-the-art neural network called Convolutional Neural Network (CNN) which performed well on MNIST dataset. A new era of modern computer vision prevailed over all the contemporary methods when AlexNet achieved a staggering 92% accuracy on ImageNet dataset.

2. Summary of the Program:

2.1 Details of the Architecture:

LeNet-5 architecture has a total of 60000 learnable parameters in the network. It has 3 convolution layers, 2 max-pooling layers, and 1 Fully connected layer followed by the output layer. The architecture is primitive compared to today's CNNs where we have millions of parameters with 100s of layers.

Figure 1 shows the architecture overview of LeNet-5 architecture.

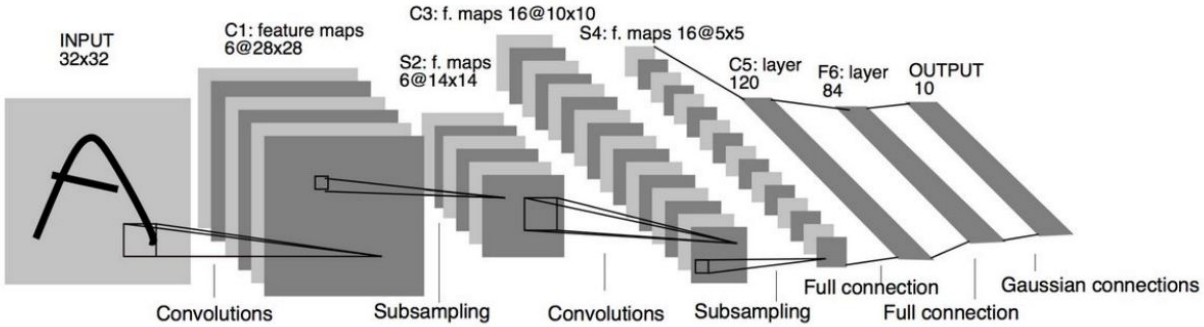


Figure 1: LeNet-5 Architecture

2.2 Code Structure:

The data required for the assignment is downloaded from the link (https://www.cs.toronto.edu/~kriz/cifar.html). LeNet5_Hw4.ipynb notebook contains code in separate cells.

2.3 Source Listing:

1. **Defining the Architecture:**

```
#Baseline Model : LeNet-5 architecture
import torch.nn as nn
import torch.nn.functional as Func
#from efficientnet_pytorch import EfficientNet

class Net(nn.Module):
 def __init__(self, model_name='efficientnet-b0'):
   super(Net, self).__init__()
   #model = EfficientNet.from_pretrained(model_name)
   #self.model = model

   #Defining 2 convolution layers
   self.convotionLayer1 = nn.Conv2d(3, 6, 5)
```

```
    self.convotionLayer2 = nn.Conv2d(6, 16, 5)
    #Defining Fully connected Layers (y = Wx + b)
    self.fullyConnected1 = nn.Linear(16 * 5 * 5, 120)
    self.fullyConnected2 = nn.Linear(120, 84)
    self.fullyConnected3 = nn.Linear(84, 10)

  def forward(self, imageFeatures):
    #imageFeatures = self.model.extract_features(imageFeatures)
                                        imageFeatures            =
Func.max_pool2d(Func.relu(self.convotionLayer1(imageFeatures))
, (2, 2))
     # If the size is a square you can only specify a single
number
                                        imageFeatures            =
Func.max_pool2d(Func.relu(self.convotionLayer2(imageFeatures))
, (2,2))
    imageFeatures = imageFeatures.view(-1, 16*5*5)
                                        imageFeatures            =
Func.relu(self.fullyConnected1(imageFeatures))
                                        imageFeatures            =
Func.relu(self.fullyConnected2(imageFeatures))
    imageFeatures = self.fullyConnected3(imageFeatures)
    return imageFeatures


cnn = Net()
print(cnn)
params = list(cnn.parameters())
print(len(params))
print(params[0].size())

input = torch.randn(3,3,32,32)
output = cnn(input)
print(output)
labels = torch.randn(10)
labels = labels.view(1, -1)
```

```
numberOfParams= sum(p.numel() for p in cnn.parameters() if
p.requires_grad)
print(numberOfParams)
```

2. **Training the network:**

```
import torch.optim as optim

def evalAccuracy():
 #Overall train and test accuracy
 correctClassification_train = 0
 totalImages_train = 0

 correctClassification_test = 0
 totalImages_test = 0

 with torch.no_grad():
   for data in trainImages:
     img, label = data
     output = cnn(img)
     for i,k in enumerate(output):
       if torch.argmax(k) == label[i]:
         correctClassification_train +=1
       totalImages_train +=1

 with torch.no_grad():
   for data in validationImages:
     img, label = data
     output = cnn(img)
     for i,k in enumerate(output):
       if torch.argmax(k) == label[i]:
         correctClassification_test +=1
       totalImages_test +=1

 trainAccuracy = correctClassification_train/totalImages_train
 testAccuracy = correctClassification_test/totalImages_test
 return trainAccuracy, testAccuracy

def training(learningRate, momentum, epochs, criterion,
optimizer):
```

```
    trainingAccuracyList = []
    testingAccuracyList = []

    running_loss = 0
    for epoch in range(epochs):
      for i, trainData in enumerate(trainImages, 0):
        imgs, labels = trainData

        #set the gradient to zero
        optimizer.zero_grad()
        outputs = cnn(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

      print(epoch, loss)

      trainAccuracy, testAccuracy = evalAccuracy()
      trainingAccuracyList.append(trainAccuracy)
      testingAccuracyList.append(testAccuracy)

    return trainingAccuracyList, testingAccuracyList

#define training parameters
learningRate = 0.001
momentum = 0.9
epochs = 80
criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(cnn.parameters(), lr=0.001,
momentum=0.9)
trainingAccuracyList, testingAccuracyList =
training(learningRate, momentum, epochs, criterion, optimizer)
print("Train Accuracy for Each Epoch: ",trainingAccuracyList)
print("Test Accuracy for Each Epoch: ", testingAccuracyList)
```

3. Test Accuracy:

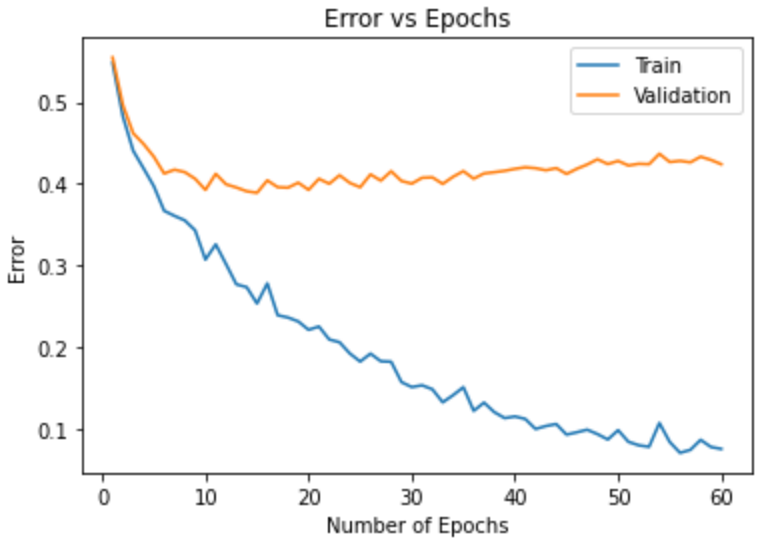
```
#Overall train and test accuracy
correctClassification = 0
totalImages = 0

with torch.no_grad():
 for data in testImages:
   img, label = data
   output = cnn(img)
   for i,k in enumerate(output):
     if torch.argmax(k) == label[i]:
       correctClassification +=1
     totalImages +=1

print ("Accuracy: ", correctClassification/totalImages)
```

4. Confusion matrix:

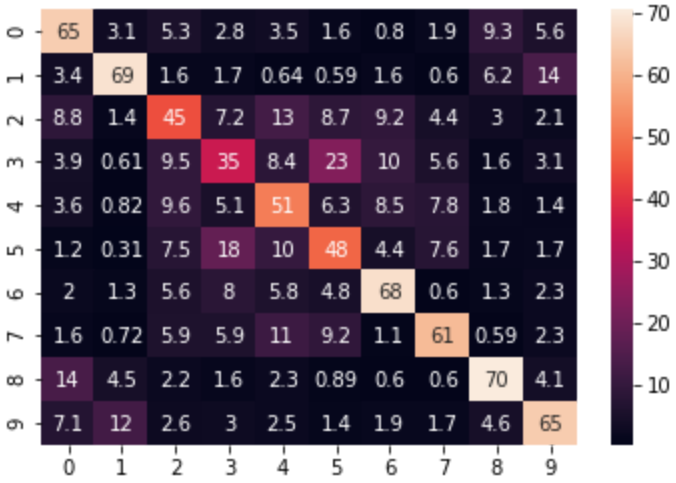
```
from sklearn.metrics import confusion_matrix
nb_classes = 10
# Initialize the prediction and label lists(tensors)
predlist=torch.zeros(0,dtype=torch.long)
lbllist=torch.zeros(0,dtype=torch.long)

with torch.no_grad():
   for i, (inputs, classes) in enumerate(testImages):
       outputs = cnn(inputs)
       _, preds = torch.max(outputs, 1)

       # Append batch prediction results
       predlist=torch.cat([predlist,preds.view(-1)])
       lbllist=torch.cat([lbllist,classes.view(-1)])
# Confusion matrix
conf_mat=confusion_matrix(lbllist.numpy(), predlist.numpy())
print(conf_mat)
# Per-class accuracy
class_accuracy=100*conf_mat.diagonal()/conf_mat.sum(1)
print(class_accuracy)
```

3. Experimental Results

1) **Baseline Model:** LeNet5 architecture
Optimizer: Adam
Criterion: Cross Entropy Loss
Epochs: 60
learning rate: 0.001
Betas: (0.9, 0.999) - Exponential decay rates
Train Accuracy: 92.44%
Validation Accuracy: 57.6%
Test Accuracy = 57.23%

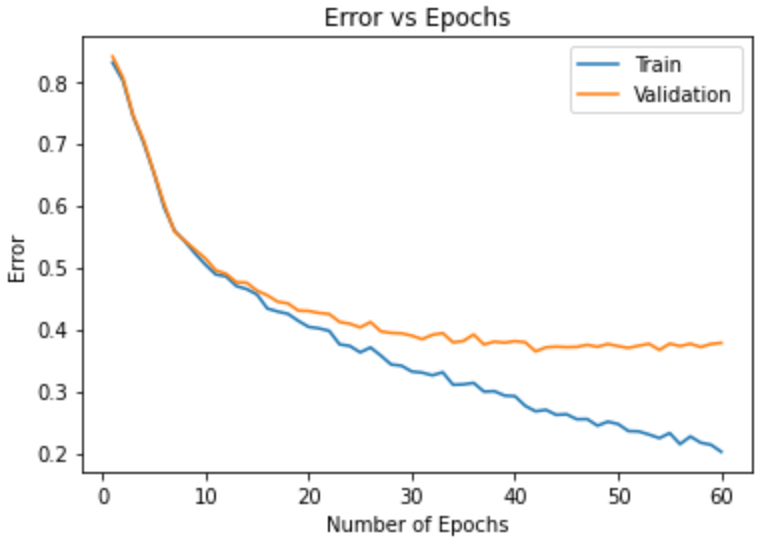


Confusion Matrix: (Diagonal elements represent class accuracy)

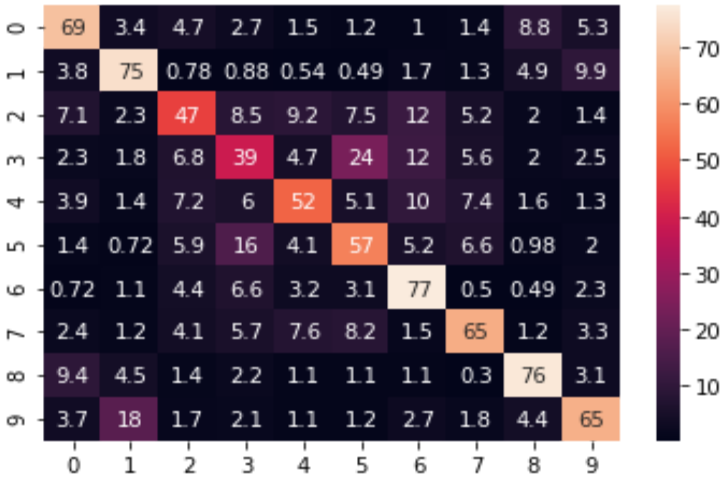


2) **Model 1:** Same as LeNet5, but SGD is used as an optimizer instead of Adam

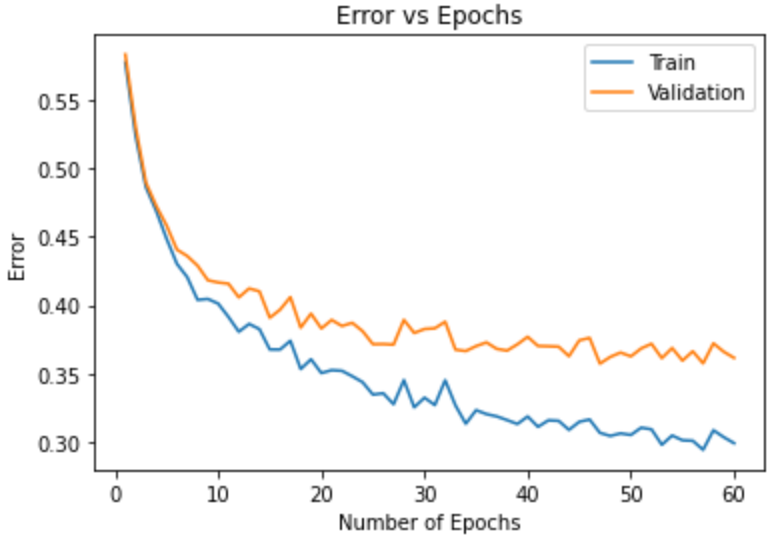
Optimizer: SGD
Criterion: Cross Entropy Loss
Epochs: 60
learning rate: 0.001
Momentum: 0.9
Model size: 62006
Train Accuracy: 79.80%
Validation Accuracy: 62.22%
Test Accuracy: 62.59%



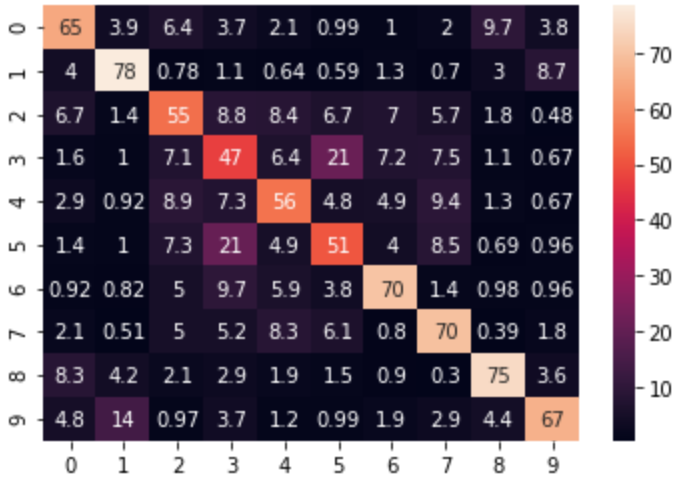
Confusion Matrix: (Diagonal elements represent class accuracy)



3) **Model 3:** Same as Model-1 but used data augmentation (Random Flip and Random erasing)
Optimizer: Adam
Criterion: Cross Entropy Loss
Epochs: 60
learning rate: 0.001
Betas: (0.9, 0.999) - Exponential decay rates
Train Accuracy: 70.09%
Validation Accuracy: 63.86%
Test Accuracy = 67.38%



Confusion Matrix: (Diagonal elements represent class accuracy)



4) **Model 4:** Reduced the size of filters from 5*5 to 3*3 with dropouts and batch-normalization.

Layers	**Parameters (Kernel Size * Channels * #Filters)**
Convolutional Layer -1 Max Pool (2*2)	3*3*3-#32
Dropout Layer with 0.3 probability (Batch Normalization)	
Convolutional Layer -2 Max Pool (2*2)	3*3*32-#64
Dropout Layer with 0.3 probability (Batch Normalization)	
Convolutional Layer -4 Max Pool (2*2)	3*3*64 - #128
Dropout Layer with 0.3 probability (Batch Normalization)	
Fully Connected Layer - 1	256
Fully Connected Layer - 2	128
Fully Connected Layer - 3	10

Optimizer: SGD

Criterion: Cross Entropy Loss

Epochs: 60

learning rate: 0.001

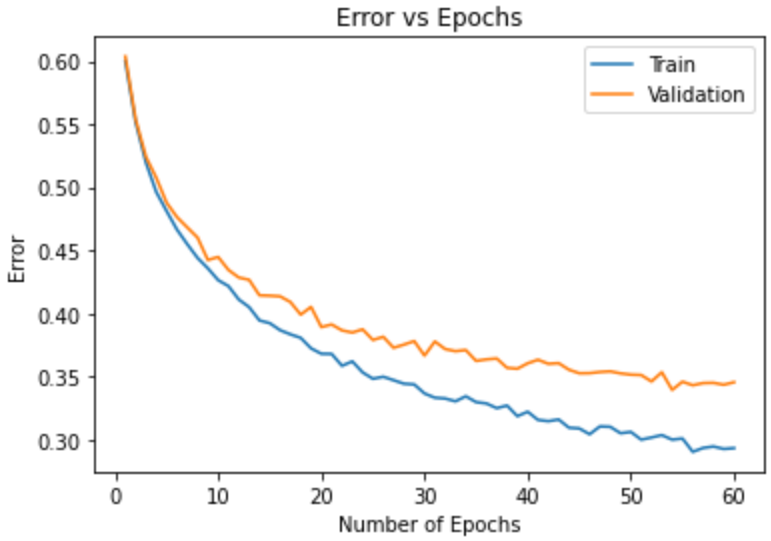
Model size: 423818

Train Accuracy: 70.68%

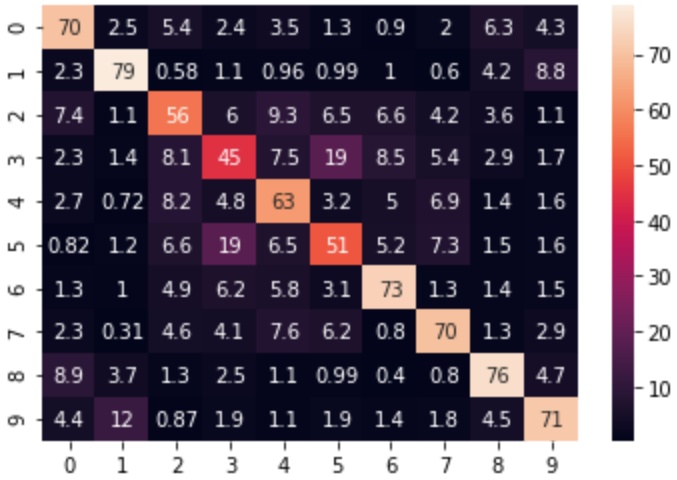
Validation Accuracy: 65.46%

Test Accuracy = 70.91%

Error vs Number of Epochs (Train and Validation set):



Confusion Matrix: (Diagonal elements represent class accuracy)



Model 5: Compressed Model by reducing the filter size from 5*5 to 3*3, Data Augmentation, Max Pool, ELU as activation, Weight Decay (0.0001), Batch Normalization, Dropouts, SGD.

Layers	**Parameters (Kernel Size * Channels * #Filters)**
Convolutional Layer -1 Max Pool (2*2)	3*3*3-#32
Dropout Layer with 0.2 probability (Batch Normalization)	
Convolutional Layer -2 Max Pool (2*2)	3*3*32-#64
Convolutional Layer -3 (Padding 1)	3*3*32 -#64
Dropout Layer with 0.3 probability (Batch Normalization)	
Convolutional Layer -4 Max Pool (2*2)	3*3*64 - #128
Convolutional Layer -5 (Padding 1)	3*3*64 - #128
Dropout Layer with 0.3 probability (Batch Normalization)	
Fully Connected Layer - 1	256
Fully Connected Layer - 2	64
Fully Connected Layer - 3	10

Optimizer: SGD

Criterion: Cross Entropy Loss

Epochs: 80

learning rate: 0.001

Activation: eLU (Exponential Linear unit)

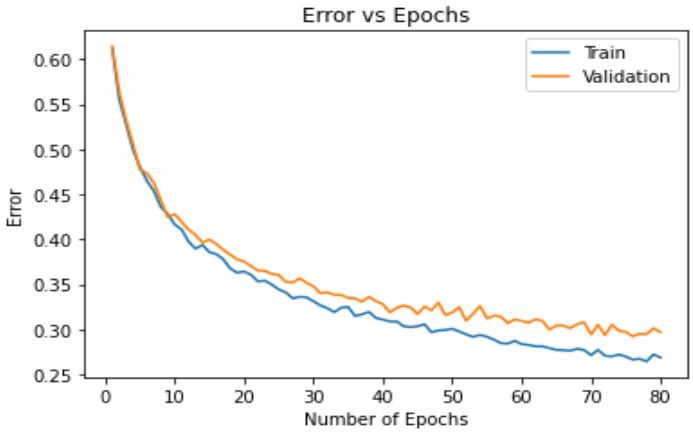
Model Size: 289130

Train Accuracy: 73.12%

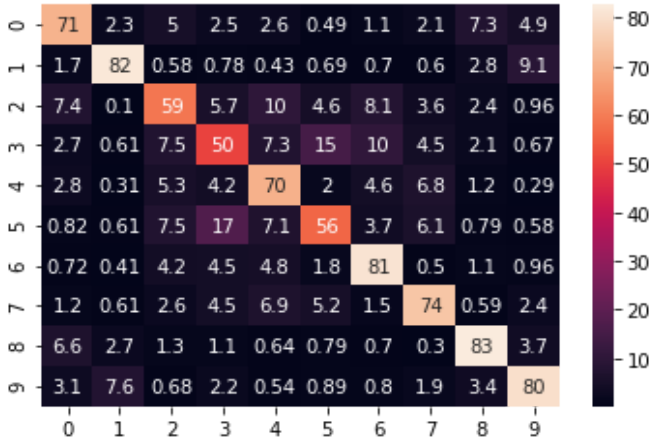
Validation Accuracy: 70.3%

Test Accuracy = 75.17% (Reason for test accuracy more than train is explained in next section)

Error vs Number of Epochs (Train and Validation set):



Confusion Matrix (Diagonal elements represent class accuracy):



4. Discussion and comments on results:

1) **Effect of Learning rate:** Decreasing the learning rate gives a smooth accuracy curve and also got maximum accuracy. Increasing the learning rate increased the training accuracy, while decreasing the accuracy on test data. This is because of overshooting problems. (results of varying learning rate is not reported here because all the experiments were performed on Baseline LeNet-5 model. Only best results are reported for each model).

2) **Change in Optimizer:** Among all the optimizers SGD has achieved highest accuracy compared to others. Adam Optimizer gave accuracy almost similar to that of SGD. Adam optimizer is an extended implementation of SGD but with adaptive change in momentum.

3) **Effect of Weight decay:** This is the addition of regularization term as we discussed in previous section. The accuracy has gone down to 10% because setting weight_decay to 1 means setting λ value to 1. This means while updating the weights, the final weight vector will be set to almost zero. So, if the model is overfitting the data then introducing a low weight_decay of 0.004 will help.

4) **Exponential Linear Unit (ELU):** D. Clevert et.al proposed a non-linear activation which can significantly help to boost the performance of deep neural networks. Similar to ReLu, Leaky ReLu, Sigmoid, ELUs avoid the vanishing gradient problem by using identity for positive values. But ELUs have pushed the mean unit activations closer to zero just like batch normalization.

$$ELU(x) = min(0, \alpha*(exp(x)-1)) + max(0, x)$$

ReLu activation is replaced by Exponential Linear Unit (ELU) in model-5 and the test accuracy has been increased from 70% to 75%

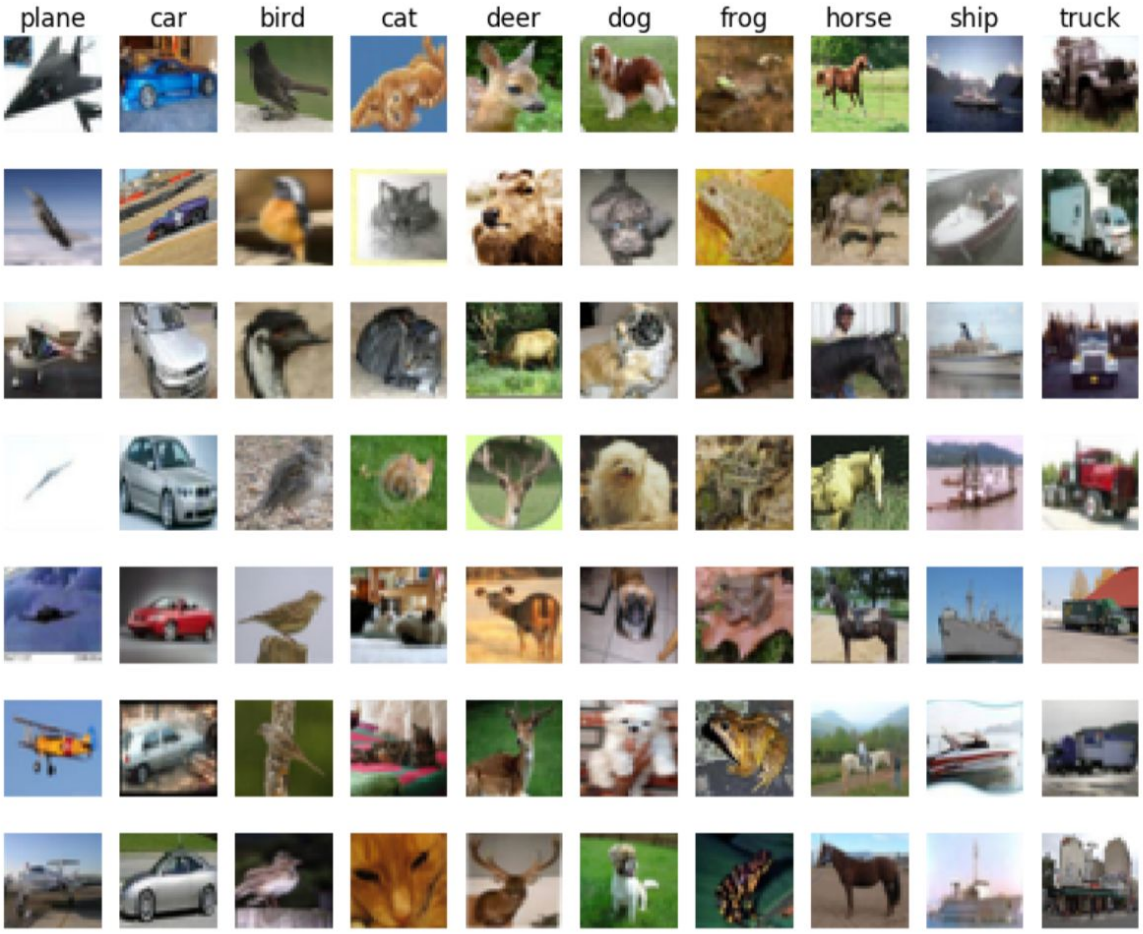
5) **Random Erasing data augmentation technique:** Usually deep learning models have hundreds and thousands of learning parameters (degrees of freedom) which is often more than the dataset we possess (constraints). When D.O.F is greater than constraints we will encounter an overfitting problem. So, in order to counterbalance this we use different data augmentation techniques. Along with commonly used data augmentation techniques like Random Flipping and cropping, a state-of-the art data augmentation technique called Random Erasing is used. In this technique a random rectangular patch is chosen in an image and the pixel values are erased with random values. This technique makes the deep learning model robust to occlusions. Employing Random erasing data augmentation technique, the test accuracy has been significantly improved from 57% to 67% in Model 3.

6) **Effect of Dropout regularization:** Even though Batch normalization induces the effect of regularization, it's always safe to add dropout layers in between to avoid overfitting. In dropout, we assign a probability to a node for being dropped. Also, we set a threshold probability to drop a node. Adding dropouts after convolution layers reduced the problem of overfitting and we can observe the effect in the plots.

7) **Why is test accuracy better than training accuracy in Model-4 and Model-5?**

Dropout forces the neural network to become a very large collection of weak classifiers. Dropout slices off some random collection of these classifiers during training. Thus, training accuracy will be restricted. Dropout turns itself off and allows all of the weak classifiers in the neural network to be used during testing. Thus, testing accuracy improves.

8) Analysis of confusion matrix:



As we can see from the confusion matrix, Class Ship and Truck have highest accuracy whereas for classes Deer and Bird the accuracy is low. Also, if we can observe carefully, both deer and dog images are being predicted as Cat. Almose 10% of the data of dog and cat each has been predicted wrongly among cats in these classes. As we can observe from the image, the resolution and also the spatial components of both the classes like ears, eyes are somewhat misleading. We can consider them as close classes. In some cases, even birds are being misclassified as cats. Around 17% of dog images are being misclassified as cats. From this we can infer that the animal classes are often misclassified among themselves rather than material classes (like ship, aeroplane etc) because of similar spatial features i.e., the feature vectors will be very close and make it harder for the classification.