EXPERIMENT 3:- DOCUMENTATION

Delhi Technological University
Shahbad Daulatpur Village, Rohini, New Delhi, Delhi, 110042

(Department of Computer science and technology)

(Deep learning: CO328 E5 Sec1 G1)

SUBMITTED BY:-
Aryan Gahlaut

2K22/CO/102

6th semester (3rd year)

SUBMITTED TO:-
Dr Anil Parihar

EXPERIMENT 3 (CIFAR-10)

Aim: To train CNN (Convolutional neural network) with different combinations of optimizers and activation function on CIFAR-10 dataset.

Dataset used: CIFAR-10

The CIFAR-10 is one of the most widely used collections in machine learning, mostly for image classification tasks. There are 60,000 images, all in color, size 32x32 pixels divided into 10 classes with 6,000 images in each class.
Training images: 50,000 and test images: 10,000. Classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Every class comprises a set of diverse images showing objects in various poses, backgrounds, and orientations. CIFAR-10 has become the popular benchmark to measure the performance of various machine learning algorithms, primarily on the field of image recognition and classification.

Platform used: Google colab

Google Colab (short for Colaboratory) is a free cloud-based notebook environment that provides users with an interactive notebook interface to write and execute Python code. It offers free access to powerful computational resources, including GPUs and TPUs, hosted by Google, to perform data analysis, machine learning, and deep learning tasks.
Colab, which integrates extremely well with Google Drive, facilitates the storage of notebooks in a cloud. Many popular Python libraries, including TensorFlow, Keras, and PyTorch are supported, meaning users can access Jupyter-like notebooks without need for any on-local setup; they can upload data from various sources, which include Google Sheets, GitHub and other cloud repositories, to use for easy collaborating and sharing with others.
One of the key features of Colab is its accessibility: since it's entirely browser-based, no installation or hardware requirements are necessary, and users can start working immediately.

Libraries and framework used:

1) os: Provides an interface that enables the developer to interact with the operating system. It involves file manipulation or path handling among other operations, and it often appears in application code as ways to work on directories or files, checking the existence of some files.

2) Torch: PyTorch is a deep learning framework developed by Facebook that is free and open source. It is highly regarded for building, training, and evaluating machine learning models, particularly neural networks. Torch supports automatic differentiation and GPU acceleration, which are highly preferred in deep learning applications.

3) torchvision: This is a companion library to PyTorch, providing datasets, model architectures, and image transformation tools that are specifically designed for computer vision tasks. It simplifies the process of working with image data and pre-trained models. tarfile: This module has the functionality for reading and writing the archives in the tar format in file packaging. It's typically used to extract compressed datasets in a .tar format, as in the example code.

3) download_url: download_url is a utility function of torchvision.datasets.utils to download a dataset from a given URL, this is handy while handling large datasets, which need to be fetched remotely.

4) random_split: A utility function in torch.utils.data that splits datasets into non-overlapping subsets. It is very often used for splitting the dataset into the training and validation set so that the model could be trained on separate data and then tested on the other set.

5)ToTensor: transformation from torchvision. transforms, this one transforms any PIL image, or any numpy array into the PyTorch tensor. Such a transformation is necessary because in most cases models in PyTorch expect data in the tensor form, it is a type of multidimensional array.

6) ImageFolder: This is a class in torchvision.datasets for loading image datasets stored in a folder structure. It assumes each folder represents a class, and the images in those folders belong to that class, so it is best suited for classification tasks.

7) matplotlib: A plotting library that is widely used in Python for visualizing data. In this context, it is used to create plots and graphs to display images or training progress, especially when working with neural networks and data visualization.

8) matplotlib.pyplot: A submodule of matplotlib that provides a MATLAB-like interface for creating plots. It is frequently used for visualizing the training process, model performance, or dataset samples during machine learning projects.

9) DataLoader: From torch.utils.data, the DataLoader class is used to load data in batches during training or testing. It manages batching, shuffling, and parallel data loading, which helps improve the performance of models by removing bottlenecks in data loading.

10) torch.nn: A module in PyTorch that provides a wide range of pre-built neural network layers and tools for building and training deep learning models. It includes layers like convolutional, fully connected, and activation layers, along with tools for defining loss functions.

11) torch.nn.functional: This sub-module contains many functions that carry out operations in tensors, ranging from activation (ReLU, sigmoid) to the loss functions: cross-entropy, and then other mathematical operation, which commonly occur during a forward pass within a neural network.

<u>Implementation</u>:

Importing necessary libraries and framework

```
# Image classification using CNN on cat vs dogs using pytorch
# CIFER 10 image classification using CNN and pytorch
# Resnet 18

# XOR and OR using single neural network
import os
import torch
import torchvision
import tarfile
from torchvision.datasets.utils import download_url
from torch.utils.data import random_split
from torchvision.transforms import ToTensor
from torchvision.datasets import ImageFolder
```

This code block imports a significant number of relevant libraries and modules for the task of deep learning.

Downloading required dataset

```
dataset_url="https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz"
download_url(dataset_url,'.')
```

The CIFAR-10 dataset is downloaded from the url link using download_url().

Extracting the data

```
with tarfile.open('./cifar10.tgz','r:gz') as tar:          # extracting
    tar.extractall(path='./data')
```

We uses the tarfile module to extract a compressed .tgz archive, specifically the CIFAR-10 dataset. The tarfile.open('./cifar10.tgz', 'r:gz') function opens the cifar10.tgz file in read mode (r) with gzip compression (gz). The with statement ensures that the file is properly opened and closed, managing resources efficiently. Inside the with block, the tar.extractall(path='./data') method extracts all the contents of the archive to the specified directory (./data).

Accessing the data and converting to tensors:

```
dataset=ImageFolder('data/cifar10/train',transform=ToTensor())
print(dataset[1])
```

We are loading the dataset within the directory, sp'data/cifar10/train' that loads the CIFAR-10 dataset. This would be the structure: subfolder = an image, belongs to this class; and so, the class transformation is made for each of them to a PyTorch tensor through some argument called transform=ToTensor() to get neural network process-ready.

The line print(dataset[1]) fetches and prints the second image of the dataset with Python's zero-based indexing. It will then print out the image data as a tensor along with its correct label. Typically, the output contains the tensor of the image with pixel values normalized between 0 and 1, as well as the class label.

Visualizing the instances of the dataset

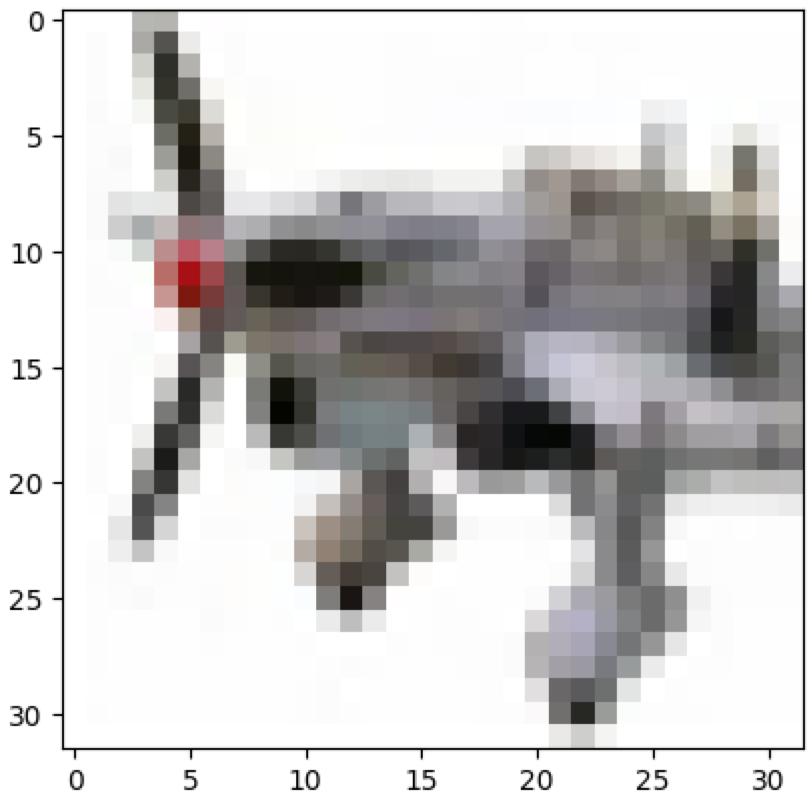
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

matplotlib.rcParams['figure.facecolor']='#ffffff'
showing(input,label)
```

This code sets up the environment for the visualization of plots in Python with the library matplotlib. It is explained briefly below as to what each part does.

import matplotlib
This imports the matplotlib library, which is the most popular library for creating static, animated, and interactive visualizations in Python.
import matplotlib.pyplot as plt: This imports the pyplot module from matplotlib, which provides a MATLAB-like interface for creating plots and charts.

%matplotlib inline: This is a special command used in Jupyter notebooks (or IPython environments) to enable inline plotting. It ensures that plots generated by matplotlib will be displayed directly within the notebook, rather than in a separate window.
matplotlib.rcParams['figure.facecolor'] = '#ffffff': This line sets the background color of the figures to white (#ffffff). rcParams is a dictionary-like object that allows customization of various default settings for matplotlib plots. In this case, it modifies the default figure background color to ensure that plots have a clean, white background.



Splitting data to train and validation data

```
train_ds , val_ds=random_split(dataset,[45000,5000])
print(len(train_ds))
print('\n')
print(len(val_ds))
```

We are splitting the dataset randomly into train_ds and val_ds having length of 45000 and 5000 respectively.
After splitting , we are printing the length of each.

Dividing the train_ds and val_ds into batches

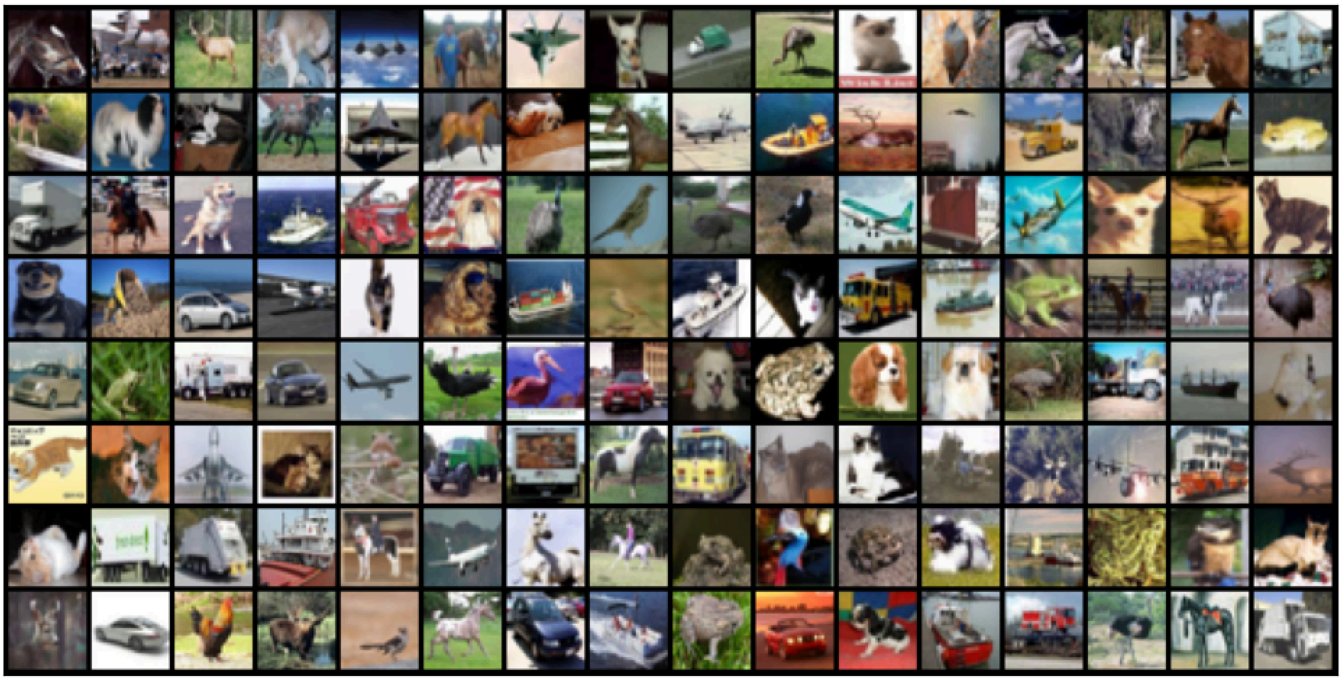
```
train_dl=DataLoader(train_ds,batch_size=128,shuffle=True,num_workers=2)      #
num worker specifies cpu cores to execute
val_dl=DataLoader(val_ds,batch_size=256,shuffle=True,num_workers=2)
```

Train_ds and val_ds are divided into series of mini batches of size of 128 and 256 respectively. Num workers represents the number of logical processors that will be used for processing and training.

Visualizing the images in the grid form

```
from torchvision.utils import make_grid
def show_batch(dl):            # to visualize the dataset at once
  for images , labels in dl:
    fig , ax=plt.subplots(figsize=(12,6))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(make_grid(images,nrow=16).permute(1,2,0)) # moving the channel
dimension to last one to make it compatible for matplotlib
    break
show_batch(train_dl)
```

This function, show_batch(dl), takes a DataLoader, plots a batch of images as one plot. The train_dl argument is supposed to be the DataLoader object and assumed to return batches of images along with the corresponding labels.
Inside the function, the for images, labels in dl: loop iterates over the DataLoader, unpacking the images and labels.ax = plt.subplots(figsize=(12, 6)) Creates a figure and axis for the plotting of images, size to be 12x6 inches. The ax.set_xticks([]) and ax.set_yticks([]) functions eliminate the x and y axis to get purely the content of the images.make_grid(images, nrow=16) uses torchvision.utils to arrange the batched-up images in a grid; nrow stands for number of images per row, it used 16. The resulting grid is then permuted with.permute(1, 2, 0) to reorder the tensor dimensions (channel, height, width) into the format expected by matplotlib (height, width, channel).
The break statement ensures that only one batch of images is shown, making the function efficient for visualizing a sample from the dataset.



Splitting test set into batches

```
test_ds=ImageFolder('data/cifar10/test',transform=ToTensor())
test_dl=DataLoader(test_ds,batch_size=128,shuffle=True,num_workers=2)
```

The test set in accessed and converted to tensors from the directory ‘data/cifar10/test’. After this , test_ds is splitted into series of mini batches of size 128 .
Num workers are set to 2 which means the number of logical processors available for processing test_dl set are two.

Defining CNN model (best one with random initialization , Adam optimizer and leakyrelu() activation function)

```
class CNN4(nn.Module):
  def __init__(self,input_shape=(3,32,32)):
    super(CNN4 , self).__init__()


self.conv1=nn.Conv2d(in_channels=3,out_channels=8,kernel_size=3,stride=1,padding
=1)

self.conv2=nn.Conv2d(in_channels=8,out_channels=16,kernel_size=3,stride=1,paddin
g=1)

self.conv3=nn.Conv2d(in_channels=16,out_channels=32,kernel_size=3,stride=1,paddi
ng=1)
```

```
    self.leakyrelu=nn.LeakyReLU()

    self.pool=nn.MaxPool2d(kernel_size=2,stride=2,padding=0)

    with torch.no_grad():
            dummy_input = torch.zeros(1, *input_shape)
            dummy_output=self._forward_conv(dummy_input)
            flattened_size = dummy_output.view(1, -1).size(1)

    self.layer1=nn.Linear(flattened_size,32)
    self.layer2=nn.Linear(32,64)
    self.layer3=nn.Linear(64,64)
    self.layer4=nn.Linear(64,64)
    self.layer5=nn.Linear(64,64)
    self.layer6=nn.Linear(64,10)

    self.dropout=nn.Dropout(0.5)

  def _forward_conv(self,x):
    x=self.pool(self.leakyrelu(self.conv1(x)))
    x=self.pool(self.leakyrelu(self.conv2(x)))
    x=self.pool(self.leakyrelu(self.conv3(x)))
    return x

  def forward(self ,x):
    x=self._forward_conv(x)
    x=torch.flatten(x,1)
    x=self.leakyrelu(self.layer1(x))
    x=self.leakyrelu(self.layer2(x))
    x=self.leakyrelu(self.layer3(x))
    x=self.leakyrelu(self.layer4(x))
    x=self.leakyrelu(self.layer5(x))
    x=self.dropout(x)
    x=self.layer6(x)

    return x

model4=CNN4()

loss_fn=nn.CrossEntropyLoss()
```

This is a CNN4 architecture as defined in nn.Module of PyTorch. The CNN4 class has been designed with the input shape (3, 32, 32), assuming a typical RGB image of size 32 x 32. The __init__ method defined three convolutional layers, namely conv1, conv2 and conv3 are nn.Conv2d followed by a Leaky ReLU activation function, which is nn.LeakyReLU.

These convolutional layers will enable the model to learn hierarchical features from the input image. After every convolutional layer in the network, a max pooling layer, nn.MaxPool2d is applied for downsampling spatial dimensions.

The forward method computes output as follows: The input goes through the convolutional layers, Leaky ReLU activations and pooling are applied. After convolutional layers, the output is flattened using torch.flatten to prepare it for fully connected layers. Fully connected layers comprise layer1, layer2, layer3, layer4, layer5, layer6, which contain the linear layers followed by Leaky ReLU activation in between, and a dropout layer (nn.Dropout(0.5)) - to prevent the overfitting, sets a fraction of input units to 0 during training; finally, the output layer, layer6 should be able to predict one out of 10 classes of the CIFAR-10 dataset.

It configures the network to use the CrossEntropyLoss function, specifically nn.CrossEntropyLoss(), as the loss criterion. This is a standard loss criterion for problems in multi-class classification. CNN4 is defined as model4, which follows this architecture as being suitable to process the CIFAR-10 dataset by way of deep convolutional layers with fully connected layers to make a prediction.

Training the CNN model

```python
opt4=torch.optim.Adam(model4.parameters(),lr=1e-3)
def train_model4(model4, loss_fn, opt4, train_dl, epoch):
    for e in range(epoch):
        model4.train()
        total_loss = 0
        for images, label in train_dl:
            opt4.zero_grad()
            output = model4(images)
            loss = loss_fn(output, label)
            loss.backward()
            opt4.step()
            total_loss += loss.item()
        print(f'Epoch {e+1}/{epoch}, Train Loss:
{total_loss/len(train_dl):.4f}')
```

This piece of code trains the CNN4 model defined above using the Adam optimizer and cross-entropy loss function. The Adam optimizer is instantiated with torch.optim.Adam(model4.parameters(), lr=1e-3), which configures it to update the model's parameters during training with a learning rate of 0.001.

The function train_model4 defines the training loop; it takes as input the model, loss function, optimizer, the data loader on which to train, and the number of epochs. Inside the function, one is iterating over the specified number of epochs using epoch. In this epoch, set up the model with model4.train(), and begin the variable total_loss defined to accumulate the total loss over all batches.

For each batch in the train_dl data loader, the optimizer gradients are cleared using opt4.zero_grad(). The model predictions are then calculated as model4(images), and the loss is calculated using the provided loss function, loss_fn. The loss is then backpropagated by calling loss.backward(), and the optimizer updates the model parameters by calling opt4.step(). The loss for the current batch is added to the total_loss variable. After passing over all the mini-batches present in the train data loader, it prints average loss for this epoch by a print statement

print(f'Epoch {e+1}/{epoch}, Train Loss: {total_loss/len(train_dl):.4f}'")

Lastly it calls the function train_model4 by passing it model4 and loss_fn followed by opt4 and train dl with epochs number set to 550, in which the program begins training. The loop running 550 iterations will adjust its parameters and returns loss value following every epoch in training.

```
Epoch 1/550, Train Loss: 1.9697
Epoch 2/550, Train Loss: 1.7288
Epoch 3/550, Train Loss: 1.6302
.
.
.
.
.
Epoch 549/550, Train Loss: 0.1473
Epoch 550/550, Train Loss: 0.1360
Epoch 550/550, Train Loss: 0.1360
```

Testing the model

```
def test_model4(model4 , test_dl , loss_fn):
  total_loss = 0
  correct = 0
  with torch.no_grad():
    for images, labels in test_dl:
      outputs = model4(images)
      loss = loss_fn(outputs , labels)
      total_loss += loss.item()
      _, predicted = torch.max(outputs, 1)
      correct += (predicted == labels).sum().item()
  print("Total loss per batch: ",total_loss/len(test_dl))
  return correct


x=test_model4(model4 , test_dl,loss_fn)
print("number of correct prediction ber batch is: ",x/len(test_dl))
print("Accuracy is: ",x/(len(test_dl)*128))
```

We are now testing the model. Correct variable will keep track number of correct predictions and total_loss will keep track of the loss through out the dataset. Gradient computation is disabled for testing.
Images are feed to network in batches and the output is then used to computed loss w.r.t to label , which is then added to toal_loss.
Predicted variable takes the class having maximum score as the prediction output.
If the predicted is equal to labels , it is correct and the correct variable is incremented by one.
Correct is then returned.
Function is called and results are printed.

Results after testing the model

```
Total loss per batch:  5.103455335279055
number of correct prediction ber batch is:  98.234352
Accuracy is:  76.645433
```

Accuracy comes out to be 76% (number of correct prediction / total number of instances).
Average number of correct prediction is 98 out of 128 instances.

Validating the model

```
def validation4(model4 , val_dl):
  correct=0
  with torch.no_grad():
    for images , labels in val_dl:
      outputs=model4(images)
      _,predicted=torch.max(outputs,1)
      print(predicted)
      print('\n')
      print(labels)
      break

validation4(model4 , val_dl)
```

The validation4 function is designed to perform validation on model4 with the validation data loader (val_dl). The function initializes the variable correct that tracks the number of correct predictions, though this is not used later in the snippet.

Inside this function, the torch.no_grad() context is used in order to prevent computation of gradients in the process of validation. It saves memory and decreases time of computations. For every batch of images and labels in the val_dl data loader, the model's outputs are calculated with model4(images).
The function torch.max(outputs, 1) is applied to get the predicted class labels by selecting the class with the highest probability for each image in the batch. The predicted labels are printed out, followed by the true labels from the dataset. The break statement will only print out the first set of predictions and labels.
This function can be very fast to inspect the model's predictions against the ground truth labels of the validation set, thus visually verifying the model behavior.

Output of validation model

```
tensor([8, 1, 2, 0, 3, 9, 9, 9, 9, 0, 9, 6, 4, 5, 3, 6, 6, 3, 2, 3, 8, 5, 6, 2,
        7, 2, 5, 8, 0, 4, 0, 2, 7, 8, 3, 3, 6, 6, 6, 5, 9, 6, 5, 6, 4, 5, 8, 1,
        2, 6, 3, 5, 7, 9, 0, 2, 1, 9, 2, 8, 7, 3, 7, 7, 4, 0, 2, 8, 4, 4, 6, 4,
        5, 3, 0, 2, 9, 8, 7, 9, 7, 4, 2, 2, 0, 7, 0, 3, 5, 7, 3, 4, 3, 7, 6, 6,
        6, 5, 4, 4, 9, 2, 8, 7, 7, 8, 7, 4, 4, 6, 4, 4, 3, 1, 1, 7, 3, 1, 5, 4,
        8, 7, 1, 9, 1, 8, 0, 4, 0, 2, 6, 7, 2, 2, 4, 8, 1, 1, 3, 0, 6, 0, 4, 0,
        0, 6, 5, 5, 0, 8, 9, 8, 2, 0, 7, 9, 8, 8, 7, 7, 2, 8, 0, 5, 0, 9, 1, 7,
        2, 9, 6, 3, 2, 5, 0, 2, 0, 5, 5, 0, 0, 5, 9, 1, 6, 9, 7, 9, 9, 3, 7, 9,
        4, 3, 0, 5, 8, 1, 8, 3, 5, 3, 7, 6, 5, 7, 3, 4, 3, 9, 6, 4, 1, 4, 7, 6,
        6, 6, 0, 8, 6, 0, 5, 9, 6, 9, 3, 0, 3, 7, 7, 7, 6, 0, 0, 3, 2, 7, 9, 8,
        8, 1, 2, 0, 9, 4, 1, 6, 0, 3, 3, 6, 9, 3, 0, 4])


tensor([8, 1, 4, 3, 2, 9, 9, 9, 9, 0, 9, 5, 4, 5, 3, 6, 6, 8, 2, 3, 8, 3, 4, 2,
        5, 2, 5, 8, 0, 4, 0, 2, 5, 8, 3, 6, 4, 6, 6, 5, 9, 6, 5, 5, 2, 4, 2, 1,
        6, 0, 3, 5, 7, 9, 0, 0, 1, 9, 2, 8, 9, 5, 7, 7, 4, 0, 2, 8, 2, 3, 2, 4,
        7, 3, 3, 2, 9, 8, 7, 9, 7, 7, 5, 6, 0, 5, 0, 2, 5, 7, 4, 7, 4, 7, 3, 2,
        6, 2, 7, 4, 9, 2, 8, 9, 7, 8, 7, 4, 2, 6, 5, 4, 7, 1, 1, 7, 4, 1, 5, 4,
        1, 7, 9, 1, 1, 8, 0, 4, 0, 2, 6, 9, 2, 2, 0, 5, 9, 1, 3, 5, 6, 0, 4, 0,
        0, 6, 5, 5, 8, 8, 9, 8, 6, 0, 7, 9, 8, 4, 0, 9, 2, 8, 0, 3, 0, 5, 1, 7,
        3, 9, 5, 5, 0, 5, 0, 4, 0, 5, 5, 8, 0, 7, 9, 1, 6, 5, 7, 9, 9, 3, 7, 2,
        4, 9, 8, 5, 8, 9, 8, 3, 5, 5, 4, 6, 5, 7, 5, 4, 5, 9, 3, 4, 1, 2, 7, 6,
        6, 6, 0, 0, 6, 0, 7, 9, 6, 1, 2, 0, 3, 5, 9, 5, 6, 0, 0, 5, 3, 7, 9, 9,
        8, 8, 5, 1, 9, 4, 1, 6, 0, 3, 6, 6, 9, 2, 0, 6])
```

<u>Conclusion</u>:

We successfully applied CNN architecture on CIFAR 10 dataset and achieved 76% accuracy.

<u>Executable file</u>(.ipynb) **<u>Cifar 10</u>**

EXPERIMENT 3 (Cats-vs-dogs)

Aim: To train CNN (Convolutional neural network) on cats-vs-dogs dataset.

Dataset used: Cats-vs-dogs (from kaggle)

The Cats vs. Dogs dataset is one of the most famous and commonly utilized binary image-classification datasets around and is ubiquitous in deep learning tutorials and competitions. It is a set of pictures of cats and dogs, mostly obtained from the web, where each picture is tagged as either "cat" or "dog."
The dataset has 25,000 images with an equal number of images of both classes, thus making it 12,500 images of cats and 12,500 images of dogs. The images in the dataset are of relatively small size. Every image is a color image with dimensions 256x256 pixels. Typically, a training set and a validation set are distinguished. Common split for the latter two sets are 80% training and 20% validation.
The main task is to construct a model that can differentiate between images of cats and dogs. This means that the model should learn features related to the shape, texture, and context that each animal belongs to. It is one of the most commonly used datasets for

computer vision and is easy for beginners and experts alike to understand because it clearly defines the goal of classification.

Platform used: Google colab

Google Colab (short for Colaboratory) is a free cloud-based notebook environment that provides users with an interactive notebook interface to write and execute Python code. It offers free access to powerful computational resources, including GPUs and TPUs, hosted by Google, to perform data analysis, machine learning, and deep learning tasks.
Colab, which integrates extremely well with Google Drive, facilitates the storage of notebooks in a cloud. Many popular Python libraries, including TensorFlow, Keras, and PyTorch are supported, meaning users can access Jupyter-like notebooks without need for any on-local setup; they can upload data from various sources, which include Google Sheets, GitHub and other cloud repositories, to use for easy collaborating and sharing with others.
One of the key features of Colab is its accessibility: since it's entirely browser-based, no installation or hardware requirements are necessary, and users can start working immediately.

Framework and libraries deployed:

1) Torch: This is the core PyTorch library and provides the functions for tensor manipulation, neural network operations, and GPU support to train models.

2) Torch.nn: the submodule of PyTorch itself, providing with pre-built layers and utilities of building and training neural networks using layers such as Conv2d, Linear or activation functions including ReLU.

3) Torch.optim: It is the module that includes optimization algorithms such as Adam, SGD, and RMSProp, which are used to update the weights of a model during training by gradients calculated.

4) Torchvision.transforms: It is a module that includes transformation utilities on images, like resizing, normalization, and tensorization of images, which is essential for pre-processing image data before feeding it into a neural network.

5) Torch.utils.data.DataLoader: Loader class is using to load batches of data to memory in data-parallel-fashion. It may support shuffling and parallel for speeding up trainings.

6) Torchvision.datasets.ImageFolder: Load dataset organized like folders, for example, there is a single folder for all images of that class. Frequently used for in-house datasets involving images categorized separately.

7) matplotlib and matplotlib.pyplot: These are the libraries that deal with visualization of data, plot generation, and other aspects related to that.
In the case of deep learning, these libraries are quite helpful in showing the images or the loss curve, model performance, while training or testing.

8) Torch.nn.functional: This submodule consists of various functions, like activation functions, which are used while applying forward pass in a neural network. The examples are ReLU and Sigmoid activation functions, cross_entropy as loss functions, etc.

Implementation:

Importing necessary libraries and mounting google drive

```
from google.colab import drive
drive.mount('/content/drive')

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import matplotlib
import matplotlib.pyplot as plt
```

Required libraries are imported after mounting google drive . Google drive is mounted in colab file to access the images data seamlessly.

Defining transform to downscale images to fixed size

```
Transform=transforms.Compose([
    transforms.Resize((96,96)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5,0.5,0.5],std=[0.5,0.5,0.5])
])
```

This code sets a collection of transformations for images through the torchvision.transforms module transforms.Compose() to preprocess input images before feeding them into a neural network.
The first transformation takes an image, resizes it uniformly to a fixed size of 96x96 pixels. This makes the input dimensions uniform since images in the dataset have different sizes. Now transforms.ToTensor() converts the image into a PyTorch tensor, which is necessary for compatibility with neural networks, as they operate on tensor data rather than other formats like PIL or NumPy arrays.
Finally, transforms.Normalize(mean = [0.5, 0.5, 0.5], std = [0.5, 0.5, 0.5]) Normalizes the pixel values of every RGB channel into a range which has a mean of 0 and a standard deviation of 1, by scaling it based on a pre-defined mean of 0.5 and pre-defined standard deviation of 0.5 on each channel for normalization, such that the centered data helps for efficient and more stable training processes. All of these transformations ensure that image data will be consistent, appropriately scaled, and in the right format to train deep learning models.

Accessing train data

```
train_data=ImageFolder(root='/content/drive/MyDrive/cats-vs-dogs/Train',transfor
m=Transform)
Train_data[0]

test_data=ImageFolder(root='/content/drive/MyDrive/cats-vs-dogs/Test',transform=
Transform)
test_data[0]
```

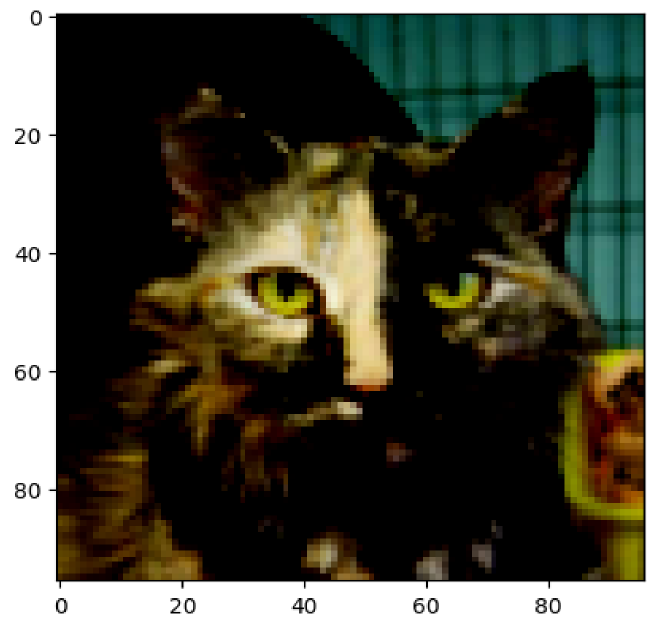
The train_data is accessed through the directory using imagefolder() and a transformation is applied to image matrices to downsales them to 3x96x96 pixels.

Visualizing the image

```
def show_image(img):
  plt.imshow(img.permute(1,2,0))                                # plotting the
image
```

```
img , label=test_data[0]
show_image(img)
```

The show_image function is to show one image out of a collection. Inside the function, the image tensor, which in the standard way has the shape of (C, H, W), where C is the number of channels- for instance if it's an RGB image that could be 3, and H and W would be the height and the width respectively--is permuted by img.permute(1, 2, 0), to get these dimensions just the way matplotlib requires to plot it: (H, W, C). The line plt.imshow(img.permute(1, 2, 0)) plots the image using matplotlib.



Splitting train and test data into series of mini batches

```
train_dl=DataLoader(train_data,batch_size=256,shuffle=True,num_workers=4,pin_mem
ory=True)              # dividing train_data into series of mini batches
test_dl=DataLoader(test_data,batch_size=256,shuffle=True,num_workers=4,pin_memor
y=True)
```

Train_data and test_data are divided into series of mini batches of fixed size of 256. Num workers represents number of logical processors which will used in performing processing task. Pin_memory set to true will reserve some memory to store the batches for training.

Defining CNN architecture

```
class CNN(nn.Module):
# CNN architecture
  def __init__(self,input_shape=(3,96,96)):
    super(CNN,self).__init__()
```

```
self.conv1=nn.Conv2d(in_channels=3,out_channels=32,kernel_size=3,stride=1,paddin
g=1)

self.conv2=nn.Conv2d(in_channels=32,out_channels=64,kernel_size=3,stride=1,paddi
ng=1)

self.conv3=nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3,stride=1,padd
ing=1)

    self.pool=nn.MaxPool2d(kernel_size=2,stride=2)

    self.leaky=nn.LeakyReLU(0.2)
    self.sigmoid=nn.Sigmoid()
    with torch.no_grad():
            dummy_input = torch.zeros(1, *input_shape)
            dummy_output = self._forward_conv(dummy_input)
            flattened_size = dummy_output.view(1, -1).size(1)

    self.layer1=nn.Linear(flattened_size,48)
    self.layer2=nn.Linear(48,48)
    self.layer3=nn.Linear(48,48)
    self.layer4=nn.Linear(48,1)

  def _forward_conv(self,x):
    x=self.pool(self.leaky(self.conv1(x)))
    x=self.pool(self.leaky(self.conv2(x)))
    x=self.pool(self.leaky(self.conv3(x)))
    return x

  def forward(self,x):
    x=self._forward_conv(x)
    x=torch.flatten(x,1)
    x=self.leaky(self.layer1(x))
    x=self.leaky(self.layer2(x))
    x=self.leaky(self.layer3(x))
    x=self.sigmoid(self.layer4(x))

    return x


model=CNN()
```
This is a CNN class, CNN, defined using PyTorch's nn.Module. This model is to work directly with input images of size 96x96 pixels with three color channels- RGB. The __init__ method sets up the layers of the network.
The network starts with three convolutional layers conv1, conv2, and conv3, each using nn.Conv2d with increasing numbers of output channels, 32, 64, and 128, respectively.

Each convolutional layer is followed by a Leaky ReLU activation (nn.LeakyReLU) and max-pooling (nn.MaxPool2d) to reduce the spatial dimensions while preserving important features.
The Leaky ReLU activation has a negative slope of 0.2. In this form, it allows a small, non-zero gradient when the input is negative, helping moderate the problem associated with vanishing gradients.
The forward pass through the convolutional layers is taken care of by the _forward_conv method, which in sequence applies convolution, Leaky ReLU activation, and pooling.
After convolutional layers, the output tensor is flattened by torch.flatten so that it gets ready for the fully connected layers. The fully connected layers: layer1, layer2, layer3, and layer4 progressively reduce the output dimensions until layer4 that produces a single output. The output of the last layer goes through a Sigmoid activation function, nn.Sigmoid, that outputs a value between 0 and 1, typically used for binary classification tasks.
model = CNN()
This architecture is designed to be used for tasks like binary classification, where the output is a single probability (e.g., whether an image belongs to one class or another).

Training the model

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Ensure
you're using the GPU
model = model.to(device)  # Move model to GPU
if torch.cuda.is_available():
  print("GPU is available!")
  print(torch.cuda.get_device_name(0))  # Prints the name of the GPU
else:
  print("GPU is not available.")
opt=torch.optim.Adam(model.parameters(),lr=1e-3)          # Adam optimiser
loss_fn=F.binary_cross_entropy
def train(model, opt, loss_fn, train_dl, epoch):
    for e in range(epoch):                                        #
Training the model
        model.train()
        total_loss = 0
        for img, label in train_dl:
            img, label = img.to(device), label.to(device)  # Move data to GPU
            img, label = img.to(torch.float32), label.to(torch.float32)  #
Ensure float labels

            opt.zero_grad()
            output = model(img).view(-1)  # Flatten output to match label shape
            loss = loss_fn(output, label)
            loss.backward()
            opt.step()
            total_loss += loss.item()

            print(f'Epoch {e+1}/{epoch}, Train Loss per batch:
{total_loss/len(train_dl):.4f}')

train(model,opt,loss_fn,train_dl,35)
```

This code snippet trains a neural network model on a GPU if it is available; otherwise, the Adam optimizer with binary cross-entropy loss. The first line sets the device to "cuda" if a GPU is available, else "cpu" by using torch.device. This model is then moved to the appropriate device, that is, the GPU or CPU using model.to(device). If a GPU exists, it is printed that one is available with the name of the GPU being torch.cuda.get_device_name(0). It then prints the message if a GPU does not exist.

It defines the optimizer with torch.optim.Adam and learns at a speed of 0.001; it also utilizes the binary cross-entropy loss function with F.binary_cross_entropy, which will be used when dealing with two-class classification.

The train method is defined such that it would specify the total number of training epochs (epoch). In an epoch, every model is assigned training mode to start training mode by model.train(). The combined loss for a total epoch during the training was summed up during this process; at each train_dl DataLoader comprising batches of image and labels for a batch; it moved both to the relevant device, using.to(device). Additionally, the data is cast to the appropriate data type (float32), as required for model processing.

The optimizer gradients are zeroed out with opt.zero_grad(), the model's output is computed for the input images. Then it passes through .view(-1) to reshape the output to be the same shape as that of the label. Then, by comparing the predicted output and actual labels, the loss is computed, which is then backpropagated into the loss with loss.backward() and the optimizer steps into the model's parameters with opt.step().

After each batch, the total loss is updated, and the average loss for the epoch is printed, so that the user can monitor the training process. Finally, the train function is called to start training the model for 35 epochs with the specified optimizer, loss function, and training data loader (train_dl).

Results of training

```
Streaming output truncated to the last 5000 lines.
Epoch 4/35, Train Loss per batch: 0.0574
Epoch 4/35, Train Loss per batch: 0.0599
Epoch 4/35, Train Loss per batch: 0.0625
Epoch 4/35, Train Loss per batch: 0.0650
.
.
.
.
Epoch 35/35, Train Loss per batch: 0.0221
Epoch 35/35, Train Loss per batch: 0.0222
Epoch 35/35, Train Loss per batch: 0.0222
```

Testing the model

```python
def test_model(model , loss_fn , test_dl):                    # Testing the model
  device = next(model.parameters()).device
  model.eval()                                              # model in evaluation mode
  with torch.no_grad():                                 # disabling gradient computation
    total_loss=0
    correct=0
    for imgs,labels in test_dl:
      imgs, labels = imgs.to(device), labels.to(device)   # Move inputs to the
same device as the model
      imgs, labels = imgs.to(torch.float32), labels.to(torch.float32)
      output=model(imgs).view(-1)
      loss=loss_fn(output,labels)
      total_loss+=loss.item()
      pred=torch.round(output)
      correct+=(pred==labels).sum().item()
    print(f'Test Loss per batch: {total_loss/len(test_dl):.4f}')
    return correct

c=test_model(model,loss_fn,test_dl)
print("Accuracy is: ",c*100/(len(test_dl)*256))
```

The `test_model` function is used to test the performance of the trained model on the test dataset. It first gets the device where the model parameters resides whether it is CPU or GPU so that the model and the data go on the same device for computation. Then it puts on the evaluation mode by calling `model.eval()`, during which some layers such as dropout switch to the appropriate inference mode. The context of the `torch.no_grad()` implements the prevention from computing gradients; this consumes most of the space in memory thus accelerating the test process because evaluation does not have gradients.

Inside the function, it iterates over the test data, taking each batch in turn, shifting the images and labels to be on the same device as the model, casting them to `float32`, and then creating predictions from these images. Finally, the total loss across all batches is added up using the binary cross-entropy `loss_fn`. The predictions are rounded to the nearest integer (0 or 1) using `torch.round(output)` and accuracy is computed by comparing these predictions with true labels. So, number of correct predictions is summed up and the function prints average loss per batch. Finally, function returns total number of correct predictions.

Then, accuracy will be calculated as correct predictions over the total number of samples in a test dataset with 256 samples per batch, and it will be printed out as a percentage. This evaluates how well the model generalizes to unseen test data-this is an overall measure of the accuracy of the model.

Results of testing the model

```
Test Loss per batch: 1.9112
Accuracy is:  80.546875
```

The test loss ber batch comes out to be **1.9112** and the accuracy (number of correct prediction / total number of instances) comes out to be **80.54%.**

Conclusion:

We have successfully trained the CNN on cats-vs-dogs dataset achieving accuracy of **80.5%.**

Executable (.ipynb) File: **Cats-vs-dogs**

Github Link: **Github link**