# Name-Devansh Yadav

Roll no-2K22/CO/159

E5-G2

Experiment 3

# CIFAR-10 Image Classification using CNN in PyTorch

## Import Required Libraries

**1. Import Required Libraries**

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

from torch.utils.data.dataloader import DataLoader

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

This section imports necessary libraries:

* torch and torchvision for deep learning and image processing.
* tarfile to extract dataset files.
* download\_url to fetch CIFAR-10 dataset.
* random\_split to divide the dataset into training and validation sets.
* ToTensor for converting images to tensors.
* ImageFolder for loading image datasets.
* DataLoader to efficiently load training and validation data.
* torch.nn, torch.nn.functional, and torch.optim for defining and optimizing the CNN model.

```python  
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  
from torch.utils.data.dataloader import DataLoader  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.optim as optim  
```

## Download and Extract CIFAR-10 Dataset

This snippet:

* Downloads the CIFAR-10 dataset from a specified URL.
* Extracts the downloaded tar file into the ./data directory.

```python  
dataset\_url = "https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz"  
download\_url(dataset\_url, '.')  
  
with tarfile.open('./cifar10.tgz', 'r:gz') as archive:  
 archive.extractall(path='./data')  
```

## Load Dataset and Split into Training & Validation Sets

**3. Load Dataset and Split into Training & Validation Sets**

train\_data = ImageFolder('data/cifar10/train', transform=ToTensor())

print(train\_data[1])

train\_set, valid\_set = random\_split(train\_data, [45000, 5000])

print(len(train\_set))

print(len(valid\_set))

* Loads CIFAR-10 images using ImageFolder and converts them into tensors.
* Splits the dataset into 45,000 training images and 5,000 validation images.
* Prints dataset sizes for verification.

```python  
train\_data = ImageFolder('data/cifar10/train', transform=ToTensor())  
train\_set, valid\_set = random\_split(train\_data, [45000, 5000])  
```

## Create Data Loaders

**4. Create Data Loaders**

train\_loader = DataLoader(train\_set, batch\_size=128, shuffle=True, num\_workers=2)

valid\_loader = DataLoader(valid\_set, batch\_size=256, shuffle=True, num\_workers=2)

* DataLoader batches the dataset for efficient processing.
* Training batch size: 128, Validation batch size: 256.
* shuffle=True ensures randomness in batches.
* num\_workers=2 allows parallel data loading for efficiency.

```python  
train\_loader = DataLoader(train\_set, batch\_size=128, shuffle=True, num\_workers=2)  
valid\_loader = DataLoader(valid\_set, batch\_size=256, shuffle=True, num\_workers=2)  
```

## Define CNN Model

**5. Define CNN Model**

class CIFAR10\_CNN(nn.Module):

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

super(CIFAR10\_CNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 32, kernel\_size=3, padding=1)

self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1)

self.conv3 = nn.Conv2d(64, 128, kernel\_size=3, padding=1)

self.fc1 = nn.Linear(128 \* 4 \* 4, 512)

self.fc2 = nn.Linear(512, 10)

def forward(self, x):

x = F.relu(F.max\_pool2d(self.conv1(x), 2))

x = F.relu(F.max\_pool2d(self.conv2(x), 2))

x = F.relu(F.max\_pool2d(self.conv3(x), 2))

x = x.view(x.size(0), -1)

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

* Defines a CNN with three convolutional layers followed by max pooling.
* Two fully connected (fc1, fc2) layers classify images into 10 categories.
* ReLU activation is used for non-linearity.
* view(x.size(0), -1) flattens the tensor before passing it to fully connected layers.

```python  
class CIFAR10\_CNN(nn.Module):  
 def \_\_init\_\_(self):  
 super(CIFAR10\_CNN, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(3, 32, kernel\_size=3, padding=1)  
 self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1)  
 self.conv3 = nn.Conv2d(64, 128, kernel\_size=3, padding=1)  
 self.fc1 = nn.Linear(128 \* 4 \* 4, 512)  
 self.fc2 = nn.Linear(512, 10)  
   
 def forward(self, x):  
 x = F.relu(F.max\_pool2d(self.conv1(x), 2))  
 x = F.relu(F.max\_pool2d(self.conv2(x), 2))  
 x = F.relu(F.max\_pool2d(self.conv3(x), 2))  
 x = x.view(x.size(0), -1)  
 x = F.relu(self.fc1(x))  
 x = self.fc2(x)  
 return x  
```

## Initialize Model, Loss Function, and Optimizer

**6. Initialize Model, Loss Function, and Optimizer**

model = CIFAR10\_CNN()

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

* Initializes CIFAR10\_CNN model.
* Uses CrossEntropyLoss for classification.
* Uses Adam optimizer with a learning rate of 0.001.

```python  
model = CIFAR10\_CNN()  
criterion = nn.CrossEntropyLoss()  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  
```

## Configure GPU Support

**7. Configure GPU Support**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

model = CIFAR10\_CNN().to(device)

* Detects if GPU is available and assigns device accordingly.
* Moves the model to the selected device (GPU if available, otherwise CPU).

```python  
device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
model = CIFAR10\_CNN().to(device)  
```

## Training Function

**8. Training Function**

def train\_model(model, train\_loader, criterion, optimizer, epochs=400):

model.train()

for epoch in range(epochs):

running\_loss = 0.0

for inputs, labels in train\_loader:

inputs, labels = inputs.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {running\_loss / len(train\_loader)}")

* Trains the model for 400 epochs.
* Moves data to GPU if available.
* Performs forward propagation, computes loss, backpropagates, and updates model weights.
* Prints loss at the end of each epoch.

```python  
def train\_model(model, train\_loader, criterion, optimizer, epochs=400):  
 model.train()  
 for epoch in range(epochs):  
 running\_loss = 0.0  
 for inputs, labels in train\_loader:  
 inputs, labels = inputs.to(device), labels.to(device)  
 optimizer.zero\_grad()  
 outputs = model(inputs)  
 loss = criterion(outputs, labels)  
 loss.backward()  
 optimizer.step()  
 running\_loss += loss.item()  
 print(f"Epoch {epoch + 1}/{epochs}, Loss: {running\_loss / len(train\_loader)}")  
```

## Validation Function

**10. Validation Function**

def validate\_model(model, valid\_loader):

model.eval()

correct = 0

total = 0

with torch.no\_grad():

for inputs, labels in valid\_loader:

inputs, labels = inputs.to(device), labels.to(device)

outputs = model(inputs)

\_, predicted = torch.max(outputs, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Validation Accuracy: {accuracy:.2f}%')

* Evaluates the model's accuracy on the validation dataset.
* Uses torch.no\_grad() to disable gradient calculation.
* Computes the percentage of correctly classified images.

```python  
def validate\_model(model, valid\_loader):  
 model.eval()  
 correct = 0  
 total = 0  
 with torch.no\_grad():  
 for inputs, labels in valid\_loader:  
 inputs, labels = inputs.to(device), labels.to(device)  
 outputs = model(inputs)  
 \_, predicted = torch.max(outputs, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
 accuracy = 100 \* correct / total  
 print(f'Validation Accuracy: {accuracy:.2f}%')  
```

## Test Function

**12. Test Function**

def test\_model(model, test\_loader):

model.eval()

correct = 0

total = 0

with torch.no\_grad():

for inputs, labels in test\_loader:

inputs, labels = inputs.to(device), labels.to(device)

outputs = model(inputs)

\_, predicted = torch.max(outputs, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Test Accuracy: {accuracy:.2f}%')

* Computes model accuracy on a test dataset.
* Similar to validation but for a separate test set.

```python  
def test\_model(model, test\_loader):  
 model.eval()  
 correct = 0  
 total = 0  
 with torch.no\_grad():  
 for inputs, labels in test\_loader:  
 inputs, labels = inputs.to(device), labels.to(device)  
 outputs = model(inputs)  
 \_, predicted = torch.max(outputs, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
 accuracy = 100 \* correct / total  
 print(f'Test Accuracy: {accuracy:.2f}%')  
```