Image Classification using CNN Architectures | Assignment

Question 1: What is a Convolutional Neural Network (CNN), and how does it differ from traditional fully connected neural networks in terms of architecture and performance on image data?

Answer:

A Convolutional Neural Network, or CNN, is a deep neural network that is specifically built for data with grid-like structures, such as images. Unlike traditional, fully connected neural networks, CNNs automatically and jointly learn spatial hierarchies of features-edges, textures, shapes, objects-through convolution operations.

How CNNs differ from fully connected neural networks:

1. Architecture:

- CNNs consist of convolutional layers, pooling layers, and sometimes batch normalization and dropout.

- These layers slide filters (kernels) over the image in order to detect patterns such as edges or textures.

- CNNs maintain spatial information from the height and width of an image.

- FNNs employ fully connected layers in which each neuron connects to every neuron in the next layer.

- They flatten the image into a 1D vector, losing spatial relationships.

2. Parameter Efficiency:

- CNNs share weights due to filters across the image. That means fewer parameters and more efficiency.

-- FNNs have large numbers of parameters because each pixel is connected to every neuron in the next layer → prone to overfitting.

3. Performance on Image Data:

- CNNs capture spatial features : edges → shapes → objects, which finally leads to:

  Higher accuracy

  Faster training

  Better generalization

- FNNs perform poorly on images because:

  They don't capture spatial relationships.

  Have huge parameter counts. Require much more data and computation.

Question 2: Discuss the architecture of LeNet-5 and explain how it laid the foundation for modern deep learning models in computer vision. Include references to its original research paper.

Answer:

LeNet-5 Architecture:

LeNet-5 was one of the earliest and most influential Convolutional Neural Network architectures, proposed by Yann LeCun et al. in the 1998 paper "Gradient-Based Learning Applied to Document Recognition," and it was designed for handwritten digit recognition on the MNIST dataset.

The network architecture consists of 7 layers excluding input, comprising convolutional layers, subsampling layers, and fully connected layers.

Layered Architecture:

1. Input Layer

- Size: 32 × 32 grayscale image
- MNIST digits are of size 28×28, zero-padded to 32×32 to preserve edge information.

2. C1 – First Convolutional Layer

- 6 filters of size 5×5
- Output: 28 × 28 × 6
- It detects simple features such as edges and corners.

3. S2 – Subsampling or Pooling Layer

- AVERAGE POOLING WITH 2×2 WINDOW, STRIDE 2
- Output: 14 × 14 × 6
- Reduces spatial dimensions and provides translation invariance.

4. C3 – Second Convolutional Layer

- 16 filters of size 5×5
- Output: 10 × 10 × 16
- Filters are not fully connected to previous maps—designed for computational efficiency.

5. S4 – Subsampling Layer

- Average pooling again (2×2, stride 2)
- 5 × 5 × 16

6. C5 – Third Convolutional Layer

- 120 filters of size 5×5
- Fully connected to S4 because the feature map is 5×5
- Output: 120

7. F6 – Fully Connected Layer

- 84 neurons
- Inspired by the biological structure of the visual cortex.

8. Output Layer

- 10-way fully connected softmax classifier
- Predicts digits 0–9

How LeNet-5 Laid the Foundation for Modern CNNs

1. Introduced the Core CNN Building Blocks

LeNet-5 pioneered the components used in today's deep learning models:

- Convolution layers
- Pooling layers
- Non-linear activations: tanh
- Fully connected layers

These components became the template for later architectures, such as AlexNet, VGG, and ResNet.

2. Demonstrated Weight Sharing & Local Receptive Fields

- It showed that local connectivity reduces parameters drastically.
- Weight sharing enhances both efficiency and generalization.

These ideas are well grounded in all modern CNNs.

3. Proved CNNs Can Outperform Traditional ML Models

The paper demonstrates that the CNN outperforms classical algorithms, such as SVMs and decision trees, on image recognition-motivating wide adoption.

4. Introduced End-to-End Learning for Vision

LeNet-5 used:

- A fully trainable architecture
- Gradient descent + backpropagation

This inspired today's end-to-end deep learning pipelines.

5. Paved the Way for Modern Architectures

Many later breakthroughs followed the design principles of LeNet-5:

- AlexNet (2012) used deeper conv layers and ReLU.
- VGGNet (2014) used stacked 3×3 filters.
- ResNet (2015) introduced skip connections, furthering stacking conv layers.

LeNet-5 thus lays the foundation for the modern era in computer vision.

Reference:

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE.

Question 3: Compare and contrast AlexNet and VGGNet in terms of design principles, number of parameters, and performance. Highlight key innovations and limitations of each.

Answer:

AlexNet vs. VGGNet – Comparison

AlexNet (2012) and VGGNet (2014) are considered landmarks in the design of CNN architectures due to their major contributions to the development of computer vision. While both achieved state-of the-art performance in ILSVRC, the two architectures differ in depth, architecture design, and design philosophy.

1. Design Principles

AlexNet (2012)

- Introduced deep convolutional networks for large-scale image classification.
- Used relatively few but large filters, such as 11×11 and 5×5.
- ReLU activation was introduced for faster training.
- Used Local Response Normalization (LRN).
- Employed overlapping max-pooling.
- Used two GPUs due to hardware limitations.

VGGNet (2014)

- Emphasized depth as the key driver of performance.
- Used very small filters of 3×3 size stacked many times instead of large filters.
- Highly uniform architecture: same filter size across layers.
- No LRN; relied on deeper layers and ReLU for effective learning.
- Cleaner and simpler design philosophy: “Deeper networks with smaller filters.”

2. Number of Parameters

Model Approx. ParametersN otes

AlexNet ~60 million Large fully connected layers contribute heavily

VGG-16, ~138 million, Much deeper (16–19 layers); parameters mainly from FC layers

VGG-19 ~ 144 million; even deeper and heavier

VGGNet is computationally more expensive since it contains more than 2× parameters compared with AlexNet.

3. Performance

Model\tILSVRC Top-5 Error\tComments

AlexNet (2012) 15.3% First deep CNN to win ImageNet

VGG-16 (2014) 7.4% Almost 2× improvement

VGG-19 (2014) ~7.3% Incremental improvement in VGG-16

Due to a much deeper structure and more refined design, VGG considerably outperformed AlexNet.

4. Key Innovations

AlexNet Innovations

- ReLU activation instead of tanh → faster training, solved vanishing gradient.
- Dropout in fully connected layers → reduced overfitting.
- Data augmentation: cropping, flipping.
- GPU parallelism (trained on dual GPUs).
- Overlapping max-pooling for better generalization.

VGGNet Innovations

- Deep architecture up to 19 layers, highlighting the power of depth.
- Small 3×3 filters that effectively increased receptive field when stacked.
- Uniform architecture, easy to generalize and reuse.
- Provided a feature extractor that is still used today for transfer learning.

5. Limitations

AlexNet Limitations

- Large filter sizes → inefficient learning.
- There is no uniformity in architecture.
- Heavy fully connected layers.
- LRN proved unnecessary in later models.

VGGNet Limitations

- Extremely large number of parameters → slow to train, heavy to deploy.
- Very expensive GPU memory requirement.
- No skip connections, unlike several of the newer models such as ResNet.
- Computationally inefficient for real-time applications.

Question 4: What is transfer learning in the context of image classification? Explain how it helps in reducing computational costs and improving model performance with limited data.

Answer:

Transfer Learning

Transfer learning means reusing a model that has been pre-trained on a large dataset, like ImageNet, as the starting point for a new, smaller dataset or task in deep learning.

Instead, we transfer learned features-edges, textures, shapes-from the pre-trained model to the new classification problem, rather than training a CNN from scratch.

Some common pre-trained models are: VGG, ResNet, Inception, MobileNet, EfficientNet, etc.

How Transfer Learning Works

There are two main approaches:

1. **Feature Extraction**
- Use the pre-trained CNN as a fixed feature extractor.
- Freeze all convolution layers.
- Replace only the final classification layers.
- Only the new fully connected layers should be trained on the small dataset.

2. Fine-Tuning

- Unfreeze some of the deeper convolution layers.
- Retrain them on the new dataset in order to adapt high-level features.
- Useful when the new dataset is a bit different from the ImageNet domain.

How Transfer Learning Reduces Computational Costs

1. **Avoid Training from Scratch**
   - Training deep CNNs from scratch requires millions of images and weeks of GPU time.
   - Pre-trained models have already learned general features.
   - You are training only a small classifier head or a few layers → much faster.

2. **Reduces Number of Trainable Parameters**
   - Most convolutional layers are frozen.
   - Only the top layers are trained.
   - This results in fewer gradient updates, less memory consumption, and reduced compute cost.

3. Requires Much Less Data

- Pre-trained CNNs learned rich feature representations.
- Works well even for small datasets (100–2000 images).
- Avoids overfitting that would result if a deep model were being trained from scratch.

How Transfer Learning Improves Model Performance

1. Better Feature Representation

- Early layers already capture generic features:

  Edges

  Curves

  Textures

- Later layers capture complex shapes and object parts.
- This leads to higher accuracy on the new dataset.

2. Helps When Data Is Limited

- Small datasets cannot train deep networks effectively.
- Transfer learning reuses knowledge learned from millions of images and improves generalization.

3. Faster Convergence

- Models converge in minutes/hours instead of days because weights start from a meaningful initialization, not random initial values.

Question 5: Describe the role of residual connections in ResNet architecture. How do they address the vanishing gradient problem in deep CNNs?

Answer:

Role of Residual Connections in ResNet

In ResNet, residual connections, also known as skip connections, enable the input of a block to skip the convolution layers and directly add to the output.

That means that the network learns a residual difference between the input and the output, instead of learning everything from scratch.

A residual block essentially conveys information forward in two ways:

1. Through standard convolution layers
2. Through a shortcut path, i.e., skip connection

How They Solve the Vanishing Gradient Problem

1. Provide a Shortcut for Gradients

In very deep networks, gradients become extremely small when backpropagated through many layers.

Skip connections give gradients a direct route to earlier layers and prevent them from disappearing.

2. Keep Valuable Information

Even if the deeper layers struggle to learn, the shortcut ensures the original information still reaches the later layers.

This maintains the stability of the network as it gets deeper.

3. Make Deep Networks Easier to Train

Instead of forcing each layer to learn a full transformation, residual connections allow layers to learn small adjustments.

This is much easier for optimization and helps the network converge faster.

4. **Avoid the "Degradation Problem"**

Before ResNet, adding more layers usually made the accuracy worse.

Residual connections fix this by ensuring that extra layers cannot harm performance—they can simply pass information forward if they don't learn anything useful.

Question 6: Implement the LeNet-5 architectures using Tensorflow or PyTorch to classify the MNIST dataset. Report the accuracy and training time.

Answer :

CODE :

```
import torch

import torch.nn as nn

import torch.optim as optim
```

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import time

# Transform MNIST to 32×32 + Tensor
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor()
])

train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

# Define LeNet-5
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.conv3 = nn.Conv2d(16, 120, 5)

        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.tanh(self.conv1(x))
        x = self.pool(x)
```

```
        x = torch.tanh(self.conv2(x))
        x = self.pool(x)
        x = torch.tanh(self.conv3(x))

        x = x.view(-1, 120)
        x = torch.tanh(self.fc1(x))
        x = self.fc2(x)
        return x

device = "cuda" if torch.cuda.is_available() else "cpu"
model = LeNet5().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train
start = time.time()
for epoch in range(10):
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

end = time.time()

# Evaluate
correct = 0
```

```
total = 0
model.eval()

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print("Test Accuracy:", accuracy)
print("Training Time:", end - start, "seconds")
```

OUTPUT :

100%|| 9.91M/9.91M [00:00<00:00, 18.0MB/s]

100%|| 28.9k/28.9k [00:00<00:00, 486kB/s]

100%|| 1.65M/1.65M [00:00<00:00, 4.53MB/s]

100%|| 4.54k/4.54k [00:00<00:00, 7.87MB/s]

Test Accuracy: 98.6

Training Time: 103.06112957000732 seconds

Question 7: Use a pre-trained VGG16 model (via transfer learning) on a small custom dataset (e.g., flowers or animals). Replace the top layers and fine-tune the model. Include your code and result discussion.

Answer :

CODE :

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
```

```
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
import os

train_dir = "/content/drive/MyDrive/cat_and_dog/train"
test_dir  = "/content/drive/MyDrive/cat_and_dog/val"

train_gen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.2,
    horizontal_flip=True
)

test_gen = ImageDataGenerator(rescale=1./255)

train_data = train_gen.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'
)

test_data = test_gen.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'
)
```

```
base_model = VGG16(
    weights='imagenet',
    include_top=False,
    input_shape=(224, 224, 3)
)

base_model.trainable = False   # freeze base model for initial training

model = models.Sequential([
    base_model,
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(train_data.num_classes, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

history = model.fit(
    train_data,
    validation_data=test_data,
    epochs=5
)
```

```
base_model.trainable = True

for layer in base_model.layers[:-4]:     # unfreeze only last 4 layers
    layer.trainable = False

model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

history_fine = model.fit(
    train_data,
    validation_data=test_data,
    epochs=5
)

plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Val')
plt.plot(history_fine.history['accuracy'], label='Train Fine-Tune')
plt.plot(history_fine.history['val_accuracy'], label='Val Fine-Tune')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("VGG16 Transfer Learning + Fine-Tuning")
plt.show()
```

OUTPUT :

Found 275 images belonging to 2 classes.

Found 70 images belonging to 2 classes.

Model: "sequential_1"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten_1 (Flatten)	(None, 25088)	0
dense_2 (Dense)	(None, 256)	6,422,784
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 2)	514

Total params: 21,137,986 (80.64 MB)

Trainable params: 6,423,298 (24.50 MB)

Non-trainable params: 14,714,688 (56.13 MB)

Epoch 1/5

9/9 **218s** 24s/step - accuracy: 0.5346 - loss: 4.1890 - val_accuracy: 0.6571 - val_loss: 1.0377

Epoch 2/5

9/9 **186s** 21s/step - accuracy: 0.6553 - loss: 1.0090 - val_accuracy: 0.7571 - val_loss: 0.5589

Epoch 3/5

9/9 **196s** 22s/step - accuracy: 0.8296 - loss: 0.4290 - val_accuracy: 0.8429 - val_loss: 0.3900

Epoch 4/5

9/9 **187s** 21s/step - accuracy: 0.8192 - loss: 0.3557 - val_accuracy: 0.8143 - val_loss: 0.3677

Epoch 5/5

9/9 **185s** 21s/step - accuracy: 0.8778 - loss: 0.2613 - val_accuracy: 0.8143 - val_loss: 0.4010

Epoch 1/5

9/9 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **230s** 26s/step - accuracy: 0.9203 - loss: 0.2273 - val_accuracy: 0.8714 - val_loss: 0.3733

Epoch 2/5

9/9 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **220s** 25s/step - accuracy: 0.9428 - loss: 0.1584 - val_accuracy: 0.9000 - val_loss: 0.3509

Epoch 3/5

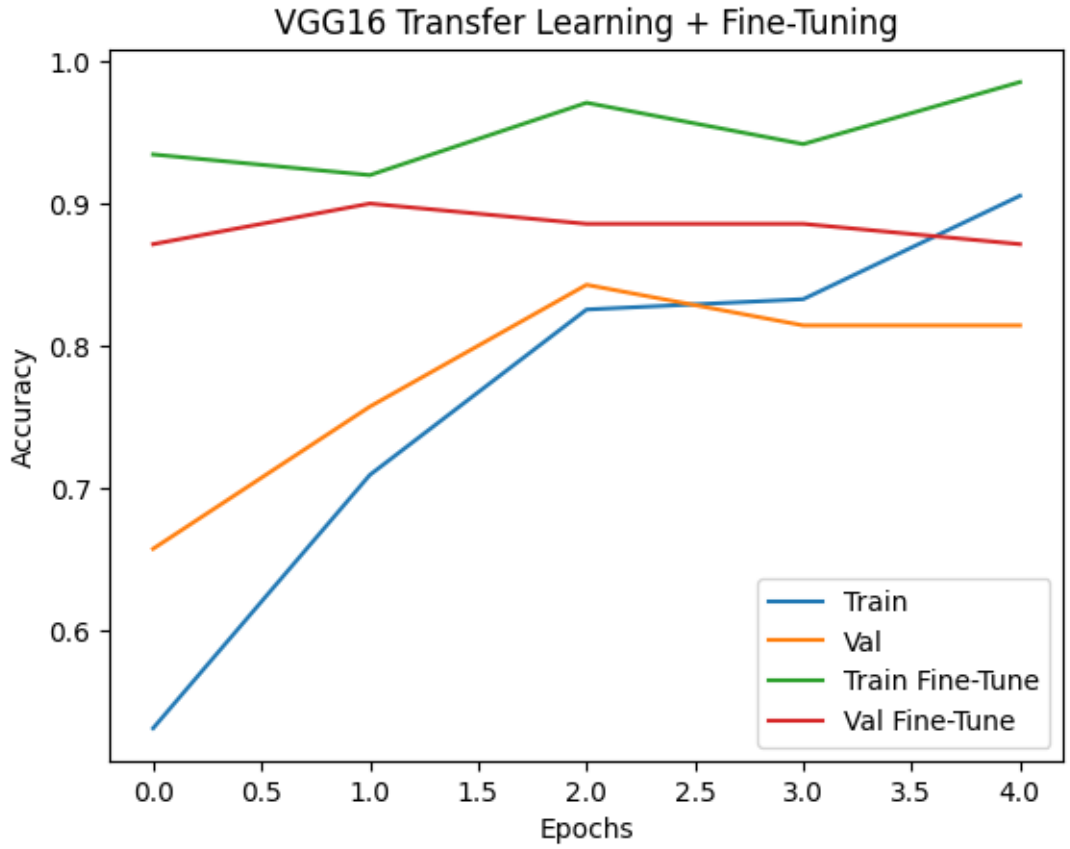
9/9 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **225s** 26s/step - accuracy: 0.9751 - loss: 0.1188 - val_accuracy: 0.8857 - val_loss: 0.3789

Epoch 4/5

9/9 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **221s** 25s/step - accuracy: 0.9484 - loss: 0.1389 - val_accuracy: 0.8857 - val_loss: 0.3776

Epoch 5/5

9/9 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **222s** 25s/step - accuracy: 0.9850 - loss: 0.0855 - val_accuracy: 0.8714 - val_loss: 0.3375



Question 8: Write a program to visualize the filters and feature maps of the first convolutional layer of AlexNet on an example input image.

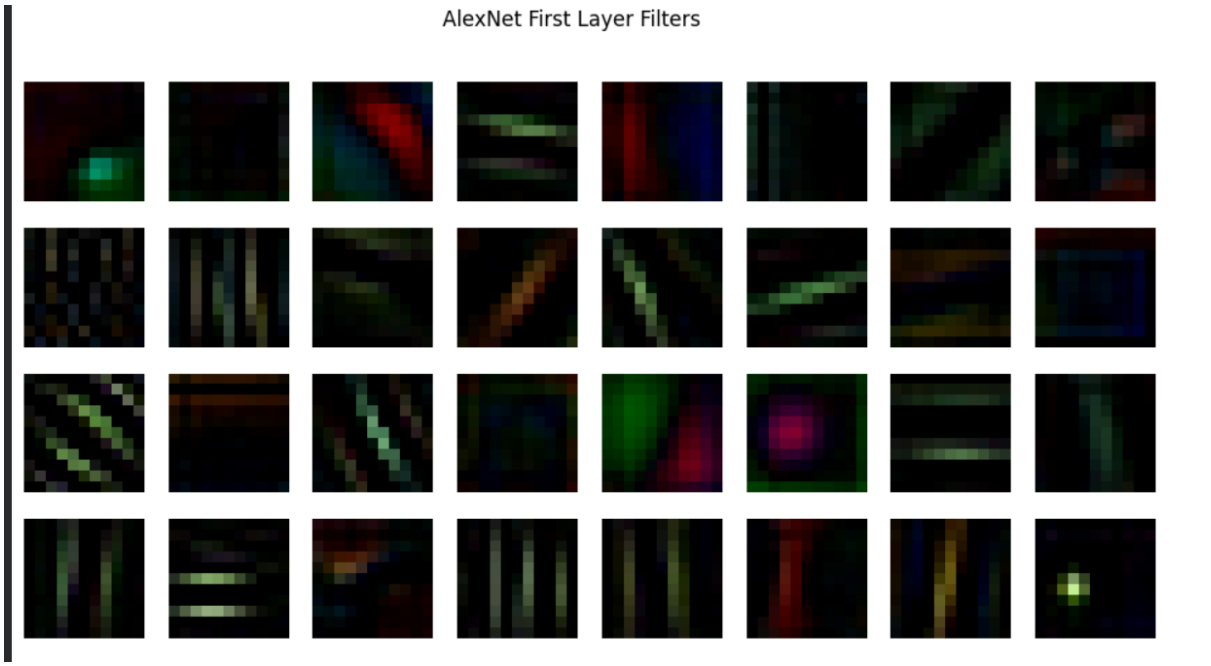
Answer :

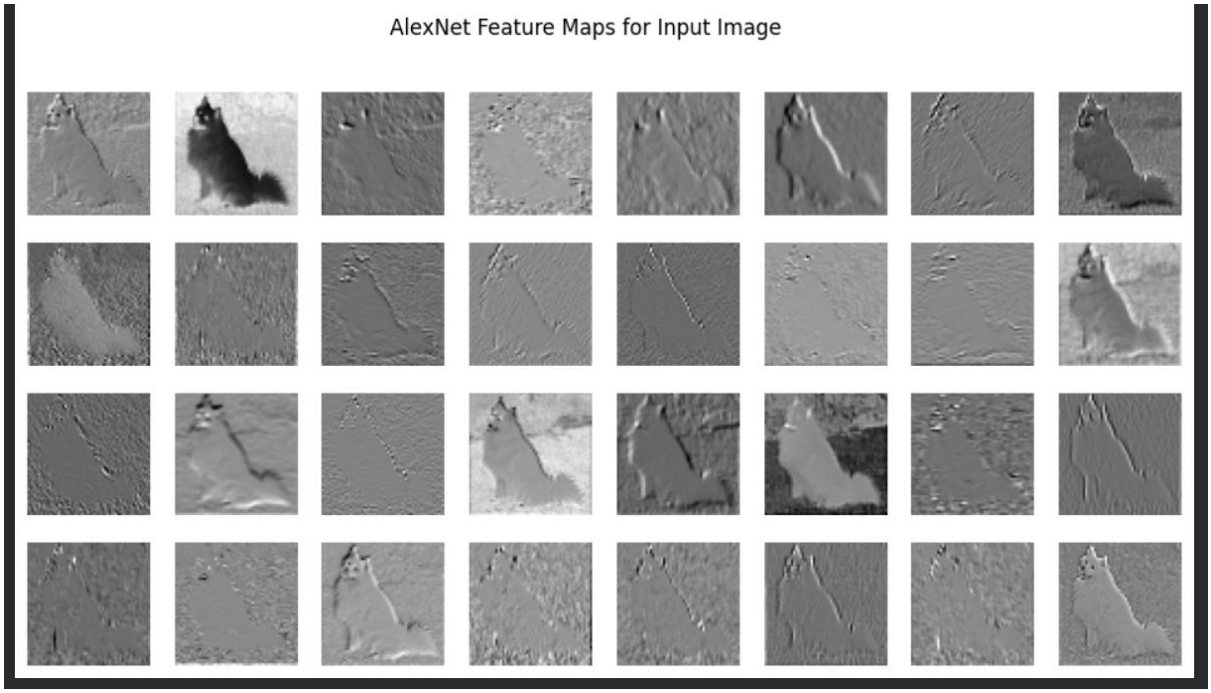
CODE :

```
import torch
import torchvision.transforms as transforms
import torchvision.models as models
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import requests
from io import BytesIO

# Load image from URL
url = "https://raw.githubusercontent.com/pytorch/hub/master/images/dog.jpg"
response = requests.get(url)
image = Image.open(BytesIO(response.content)).convert("RGB")

transform = transforms.Compose([
    transforms.Resize((227,227)),
    transforms.ToTensor()
])

input_img = transform(image).unsqueeze(0)

# Load AlexNet
alexnet = models.alexnet(weights="IMAGENET1K_V1")
```

```
features = alexnet.features[0]  # First conv layer

#  Visualize Filters
w = features.weight.data.clone()

fig, ax = plt.subplots(4,8, figsize=(12,6))
for i in range(32):
    ax[i//8][i%8].imshow(w[i].permute(1,2,0))
    ax[i//8][i%8].axis("off")
plt.suptitle("AlexNet First Layer Filters")
plt.show()

# Visualize Feature Maps
with torch.no_grad():
    fmap = features(input_img)

fmap = fmap.squeeze().cpu().numpy()

fig, ax = plt.subplots(4,8, figsize=(12,6))
for i in range(32):
    ax[i//8][i%8].imshow(fmap[i], cmap="gray")
    ax[i//8][i%8].axis("off")
plt.suptitle("AlexNet Feature Maps for Input Image")
plt.show()
```

OUTPUT :


AlexNet First Layer Filters


AlexNet Feature Maps for Input Image

Question 9: Train a GoogLeNet (Inception v1) or its variant using a standard dataset like CIFAR-10. Plot the training and validation accuracy over epochs and analyze overfitting or underfitting.

Answer :

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.models import googlenet
import matplotlib.pyplot as plt


# 1. Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# 2. CIFAR-10 preprocessing
transform = transforms.Compose([
    transforms.Resize((224, 224)),      # GoogLeNet expects 224x224
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))
])


trainset = torchvision.datasets.CIFAR10(root="./data", train=True, download=True,
transform=transform)
testset  = torchvision.datasets.CIFAR10(root="./data", train=False, download=True,
transform=transform)


trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader  = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
# 3. Load Pretrained GoogLeNet
net = googlenet(weights="IMAGENET1K_V1")
net.fc = nn.Linear(1024, 10)   # CIFAR-10 has 10 classes
net = net.to(device)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0001)

# 4. Train Loop
train_acc_list = []
val_acc_list = []

def accuracy(loader):
    net.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = net(x)
            _, preds = torch.max(outputs, 1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100 * correct / total



epochs = 10
for epoch in range(epochs):
    net.train()
    running_loss = 0

    for images, labels in trainloader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = net(images)
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # Record accuracy
    train_acc = accuracy(trainloader)
    val_acc   = accuracy(testloader)

    train_acc_list.append(train_acc)
    val_acc_list.append(val_acc)

    print(f"Epoch {epoch+1}/{epochs} - Train Acc: {train_acc:.2f}%  Val Acc: {val_acc:.2f}%")

# 5. Plot Accuracy Curves
plt.figure(figsize=(8,5))
plt.plot(train_acc_list, label="Train Accuracy")
plt.plot(val_acc_list, label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
plt.title("GoogLeNet Training vs Validation Accuracy on CIFAR-10")
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT :

100%|| 170M/170M [02:42<00:00, 1.05MB/s]

Downloading: "https://download.pytorch.org/models/googlenet-1378be20.pth" to /root/.cache/torch/hub/checkpoints/googlenet-1378be20.pth

100%|| 49.7M/49.7M [00:00<00:00, 184MB/s]

Epoch 1/10 - Train Acc: 96.93% Val Acc: 93.60%

Epoch 2/10 - Train Acc: 98.95% Val Acc: 94.31%

Epoch 3/10 - Train Acc: 99.60% Val Acc: 94.88%

Epoch 4/10 - Train Acc: 99.71% Val Acc: 94.75%

Epoch 5/10 - Train Acc: 99.43% Val Acc: 94.01%

Epoch 5/10 - Train Acc: 99.43% Val Acc: 94.01%

Epoch 6/10 - Train Acc: 99.55% Val Acc: 94.31%

Epoch 6/10 - Train Acc: 99.55% Val Acc: 94.31%

Epoch 7/10 - Train Acc: 99.85% Val Acc: 94.99%

Epoch 7/10 - Train Acc: 99.85% Val Acc: 94.99%

Epoch 8/10 - Train Acc: 99.76% Val Acc: 94.47%

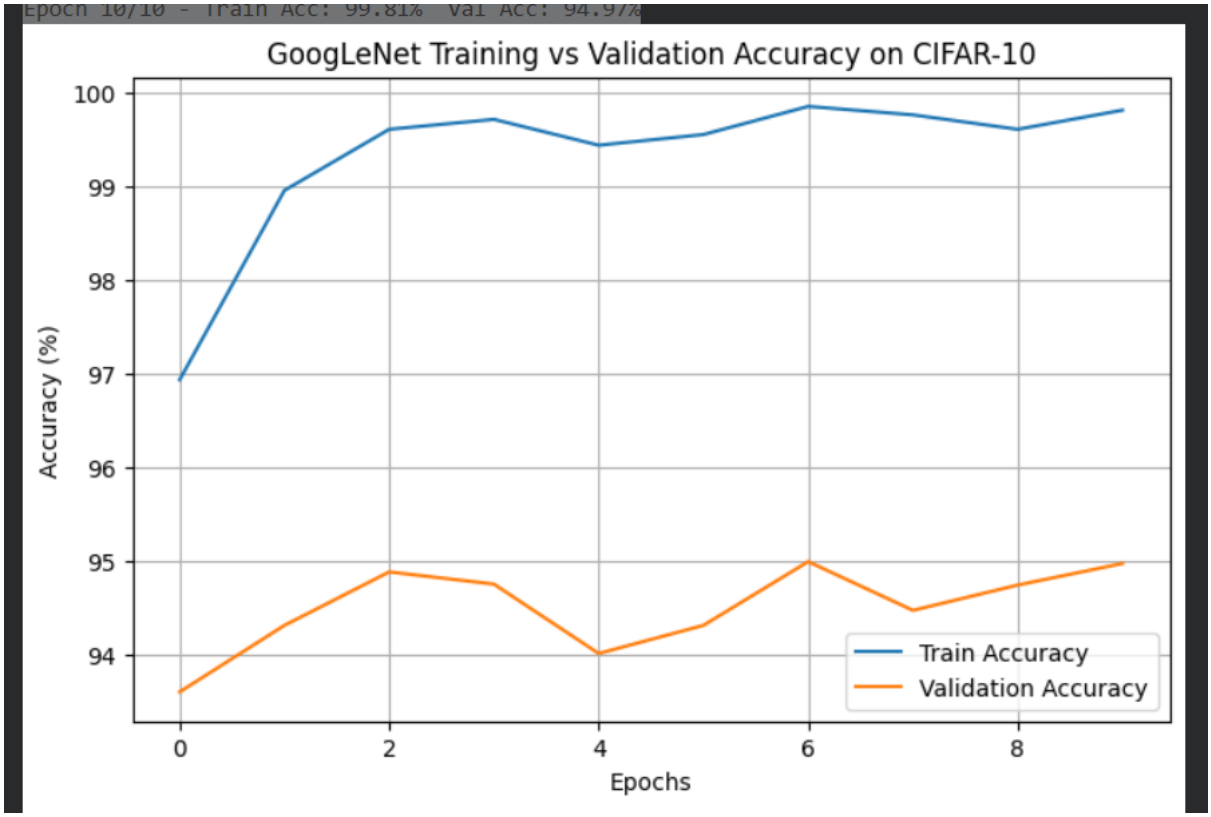
Epoch 8/10 - Train Acc: 99.76% Val Acc: 94.47%

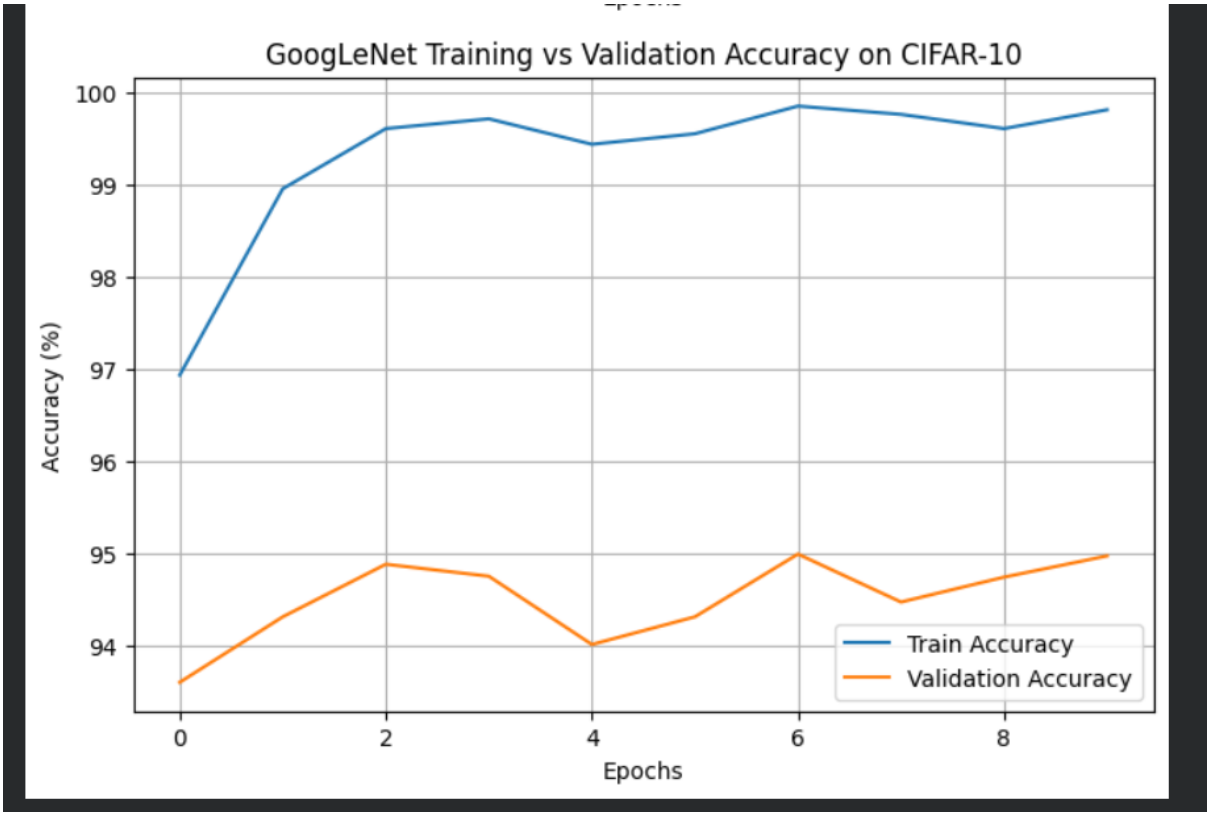
Epoch 9/10 - Train Acc: 99.60% Val Acc: 94.74%

Epoch 9/10 - Train Acc: 99.60% Val Acc: 94.74%

Epoch 10/10 - Train Acc: 99.81% Val Acc: 94.97%

Epoch 10/10 - Train Acc: 99.81% Val Acc: 94.97%





Question 10: You are working in a healthcare AI startup. Your team is tasked with developing a system that automatically classifies medical X-ray images into normal, pneumonia, and COVID-19. Due to limited labeled data, what approach would you suggest using among CNN architectures discussed (e.g., transfer learning with ResNet or Inception variants)? Justify your approach and outline a deployment strategy for production use.

Answer :

My Approach

DenseNet121 / ResNet50 are commonly used and achieve strong performance on CXR COVID/pneumonia tasks in published work.

- **Dense connections reuse features**, improving gradient flow → excellent on small datasets.
- **Very strong performance on chest X-ray tasks** (NIH, COVID datasets).
- **Fewer parameters** than ResNet → reduces overfitting.
- Extracts **fine-grained local features**, important for pneumonia/COVID opacities.

Model Development Strategy

Data Preparation

- Normalize using ImageNet preprocessing.
- Aggressive augmentations to increase dataset size:
    - rotations (10–15°)

    - width/height shifts
    - zoom
    - horizontal flip
- Split:
    - 70% train
    - 15% validation
    - 15% test

Training with Transfer Learning

Stage 1: Freeze the DenseNet backbone

- Train only the classifier head.
- Benefits:
    - Prevents overfitting early
    - Learns dataset-specific patterns slowly

Stage 2: Fine-tune last 20–40 layers

- Unfreeze deeper layers to learn X-ray patterns:
    - Ground glass opacities (COVID)
    - Consolidation (Pneumonia)
    - Clear lungs (Normal)

Callbacks to prevent overfitting

- EarlyStopping
- ReduceLROnPlateau
- ModelCheckpoint

Backend Deployement

Streamlit

- Create an API endpoint:
    - /predict
- Upload X-ray as image
- API returns:
    - prediction
    - confidence

CODE :

```
import os

import numpy as np

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.applications import DenseNet121

from tensorflow.keras.applications.densenet import preprocess_input

from tensorflow.keras import layers, Model

from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
```

```
DATA_DIR = "/content/drive/MyDrive/Xray"   # change to your path
IMG_SIZE = (224, 224)
BATCH_SIZE = 16
NUM_CLASSES = 3

# ------------------------------
# 1. Data Generators
# ------------------------------
train_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True
)

val_gen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_loader = train_gen.flow_from_directory(
    os.path.join(DATA_DIR, "train"),
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    shuffle=True
)

val_loader = val_gen.flow_from_directory(
    os.path.join(DATA_DIR, "val"),
    target_size=IMG_SIZE,
```

```
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    shuffle=False
)

# ------------------------------
# 2. Build Model (DenseNet121)
# ------------------------------
base = DenseNet121(weights="imagenet", include_top=False,
                   input_shape=(224, 224, 3))

base.trainable = False  # freeze backbone

x = layers.GlobalAveragePooling2D()(base.output)
x = layers.Dropout(0.4)(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.3)(x)
output = layers.Dense(NUM_CLASSES, activation="softmax")(x)

model = Model(inputs=base.input, outputs=output)
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

model.summary()

# ------------------------------
# 3. Callbacks
# ------------------------------
```

```
callbacks = [
    ModelCheckpoint("model_top.h5", save_best_only=True, monitor="val_accuracy"),
    ReduceLROnPlateau(monitor="val_loss", patience=3, factor=0.5),
    EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)
]

# ------------------------------
# 4. Train only the top layers
# ------------------------------
history1 = model.fit(
    train_loader,
    validation_data=val_loader,
    epochs=8,
    callbacks=callbacks
)

# ------------------------------
# 5. Fine-tune: unfreeze last layers
# ------------------------------
for layer in base.layers[-40:]:
    layer.trainable = True

model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

history2 = model.fit(
    train_loader,
    validation_data=val_loader,
```

```
    epochs=6,
    callbacks=callbacks
)

# ------------------------------
# 6. Save final model
# ------------------------------
model.save("cxr_densenet_model")
```

OUTPUT :

Total params: 7,300,675 (27.85 MB)

Trainable params: 263,171 (1.00 MB)

Non-trainable params: 7,037,504 (26.85 MB)

/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.

  self._warn_if_super_not_called()

Epoch 1/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **0s** 7s/step - accuracy: 0.4218 - loss: 1.4689

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **686s** 8s/step - accuracy: 0.4229 - loss: 1.4655 - val_accuracy: 0.9000 - val_loss: 0.3892 - learning_rate: 1.0000e-04

Epoch 2/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **247s** 3s/step - accuracy: 0.6596 - loss: 0.7766 - val_accuracy: 0.8952 - val_loss: 0.3111 - learning_rate: 1.0000e-04

Epoch 3/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **245s** 3s/step - accuracy: 0.7883 - loss: 0.5532 - val_accuracy: 0.8810 - val_loss: 0.3101 - learning_rate: 1.0000e-04

Epoch 4/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **0s** 3s/step - accuracy: 0.8149 - loss: 0.4365

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **248s** 3s/step - accuracy: 0.8148 - loss: 0.4368 - val_accuracy: 0.9238 - val_loss: 0.2537 - learning_rate: 1.0000e-04

Epoch 5/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **266s** 3s/step - accuracy: 0.8018 - loss: 0.4506 - val_accuracy: 0.9190 - val_loss: 0.2690 - learning_rate: 1.0000e-04

Epoch 6/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **253s** 3s/step - accuracy: 0.8653 - loss: 0.3832 - val_accuracy: 0.9238 - val_loss: 0.2358 - learning_rate: 1.0000e-04

Epoch 7/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **243s** 3s/step - accuracy: 0.8524 - loss: 0.3714 - val_accuracy: 0.9000 - val_loss: 0.2798 - learning_rate: 1.0000e-04

Epoch 8/8

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **265s** 3s/step - accuracy: 0.8678 - loss: 0.3456 - val_accuracy: 0.9143 - val_loss: 0.2671 - learning_rate: 1.0000e-04

Epoch 1/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **0s** 3s/step - accuracy: 0.5954 - loss: 1.0788

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **293s** 3s/step - accuracy: 0.5962 - loss: 1.0769 - val_accuracy: 0.9333 - val_loss: 0.2434 - learning_rate: 1.0000e-05

Epoch 2/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **306s** 3s/step - accuracy: 0.7901 - loss: 0.5746 - val_accuracy: 0.9190 - val_loss: 0.2603 - learning_rate: 1.0000e-05

Epoch 3/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **249s** 3s/step - accuracy: 0.8393 - loss: 0.4144 - val_accuracy: 0.9190 - val_loss: 0.2638 - learning_rate: 1.0000e-05

Epoch 4/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **254s** 3s/step - accuracy: 0.8500 - loss: 0.3718 - val_accuracy: 0.9143 - val_loss: 0.2631 - learning_rate: 1.0000e-05

Epoch 5/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **258s** 3s/step - accuracy: 0.8518 - loss: 0.3669 - val_accuracy: 0.9190 - val_loss: 0.2549 - learning_rate: 5.0000e-06

Epoch 6/6

81/81 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **257s** 3s/step - accuracy: 0.8844 - loss: 0.3190 - val_accuracy: 0.9190 - val_loss: 0.2549 - learning_rate: 5.0000e-06

Deployment code :

```
%%writefile app.py

import streamlit as st

import numpy as np

import tensorflow as tf

from tensorflow.keras.preprocessing import image

from PIL import Image


# --------------------------------

# Load Model

# --------------------------------

MODEL_PATH = "cxr_densenet_model.keras"


@st.cache_resource

def load_model():

    model = tf.keras.models.load_model(MODEL_PATH)

    return model


model = load_model()


# --------------------------------

# App Title

# --------------------------------
```

```
st.title("Medical X-ray Classifier (Normal / Pneumonia / COVID-19)")
st.write("Upload a chest X-ray image to classify using a fine-tuned ResNet50 model.")

# ---------------------------------
# Image Upload
# ---------------------------------
uploaded_file = st.file_uploader("Upload Chest X-ray (JPEG/PNG)", type=["jpg", "jpeg", "png"])

class_names = ["Normal", "Pneumonia", "COVID-19"]

# ---------------------------------
# Prediction Function
# ---------------------------------
def preprocess_image(img):
    img = img.resize((224, 224))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = tf.keras.applications.resnet50.preprocess_input(img_array)
    return img_array

def predict(img):
    processed = preprocess_image(img)
    preds = model.predict(processed)
    class_id = np.argmax(preds)
    confidence = preds[0]
    return class_id, confidence

# ---------------------------------
# Run Prediction
# ---------------------------------
if uploaded_file is not None:
```

```
    img = Image.open(uploaded_file).convert("RGB")

    st.image(img, caption="Uploaded X-ray", use_column_width=True)

    st.write(" Making prediction...")
    class_id, confidence = predict(img)

    st.write(f"## 🩺 Prediction: **{class_names[class_id]}**")

    # Confidence Bar
    st.write("### Confidence Scores")
    st.bar_chart({
        "Confidence": {
            "Normal": float(confidence[0]),
            "Pneumonia": float(confidence[1]),
            "COVID-19": float(confidence[2])
        }
    })
```

OUTPUT :

Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.

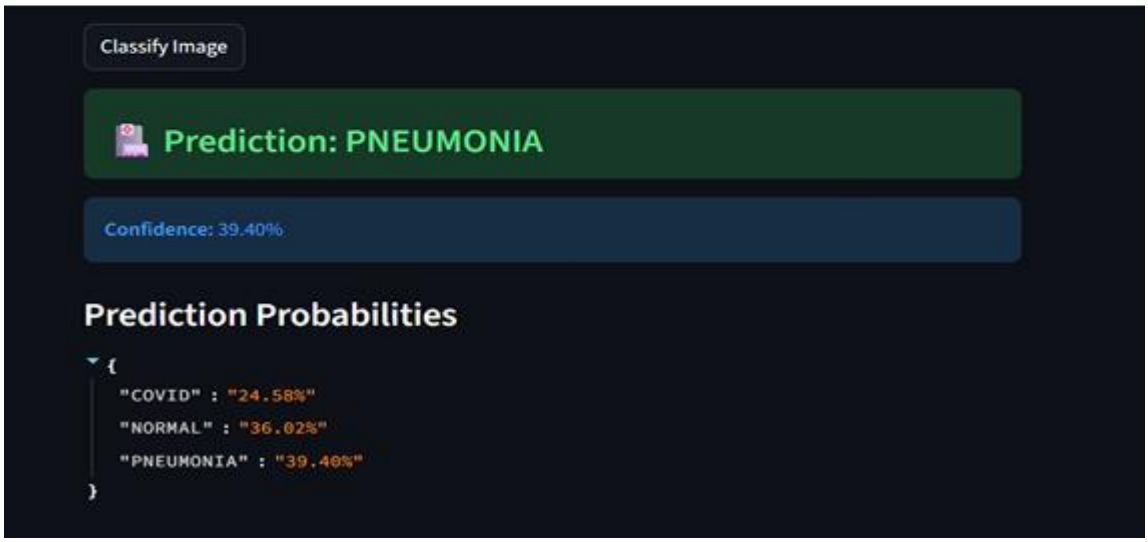
You can now view your Streamlit app in your browser.

Local URL: **http://localhost:8501**

Network URL: **http://172.28.0.12:8501**

External URL: **http://104.199.237.16:8501**


Upload X-ray Image
Drag and drop file here
Limit 200MB per file • JPG, JPEG, PNG
Browse files
PNEUMONIA_915.png 52.7KB
R
Uploaded Image


Classify Image
Prediction: PNEUMONIA
Confidence: 39.40%
Prediction Probabilities
{
"COVID" : "24.58%"
"NORMAL" : "36.02%"
"PNEUMONIA" : "39.40%"
}