```
Start coding or generate with AI.
```

```
# 1. Fresh install kaggle
!pip uninstall -y kaggle
!pip install --upgrade kaggle

# 2. Upload kaggle.json
from google.colab import files
files.upload()  # upload kaggle.json

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

# 3. Download & unzip new dataset
!rm -rf TB_Chest_Xray
!kaggle datasets download -d tawsifurrahman/tuberculosis-tb-chest-xray-dataset
!unzip -q tuberculosis-tb-chest-xray-dataset.zip -d ./TB_Chest_Xray

# 4. Check folder structure
!ls ./TB_Chest_Xray/Tuberculosis_Chest_Radiography_Database

```

```
Found existing installation: kaggle 1.7.4.5
Uninstalling kaggle-1.7.4.5:
  Successfully uninstalled kaggle-1.7.4.5
Collecting kaggle
  Downloading kaggle-1.7.4.5-py3-none-any.whl.metadata (16 kB)
Requirement already satisfied: bleach in /usr/local/lib/python3.12/dist-packages (from kaggle) (6.2.0)
Requirement already satisfied: certifi>=14.05.14 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2025.8.3)
Requirement already satisfied: charset-normalizer in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.4.3)
Requirement already satisfied: idna in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.10)
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from kaggle) (5.29.5)
Requirement already satisfied: python-dateutil>=2.5.3 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.9.0.post0)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.12/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.32.4)
Requirement already satisfied: setuptools>=21.0.0 in /usr/local/lib/python3.12/dist-packages (from kaggle) (75.2.0)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.17.0)
Requirement already satisfied: text-unidecode in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from kaggle) (4.67.1)
Requirement already satisfied: urllib3>=1.15.1 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.5.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.12/dist-packages (from kaggle) (0.5.1)
Downloading kaggle-1.7.4.5-py3-none-any.whl (181 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 181.2/181.2 kB 15.0 MB/s eta 0:00:00
Installing collected packages: kaggle
Successfully installed kaggle-1.7.4.5
 Choose Files  No file chosen    Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving kaggle.json to kaggle.json
Dataset URL: https://www.kaggle.com/datasets/tawsifurrahman/tuberculosis-tb-chest-xray-dataset
License(s): copyright-authors
Downloading tuberculosis-tb-chest-xray-dataset.zip to /content
 98% 650M/663M [00:03<00:00, 119MB/s]
100% 663M/663M [00:06<00:00, 100MB/s]
ls: cannot access './TB_Chest_Xray/Tuberculosis_Chest_Radiography_Database': No such file or directory
```

```
import os, shutil
from sklearn.model_selection import train_test_split

data_dir = "./TB_Chest_Xray/TB_Chest_Radiography_Database"
split_dir = "./TB_SPLIT"
os.makedirs(split_dir, exist_ok=True)

for cls in ["Normal", "Tuberculosis"]:
    cls_dir = os.path.join(data_dir, cls)
    images = [os.path.join(cls_dir, f) for f in os.listdir(cls_dir) if f.lower().endswith((".png", "
    # First split: train vs temp (val+test)
    train, temp = train_test_split(images, test_size=0.2, random_state=42, stratify=[cls]*len(images)
    # Second split: val vs test
    val, test = train_test_split(temp, test_size=0.5, random_state=42, stratify=[cls]*len(temp))
```

```
    for split, split_files in zip(["train", "val", "test"], [train, val, test]):
        split_path = os.path.join(split_dir, split, cls)
        os.makedirs(split_path, exist_ok=True)
        for img in split_files:
            shutil.copy(img, split_path)

print("Dataset split created at:", split_dir)

```

```
Dataset split created at: ./TB_SPLIT
```

```
import torch
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import torchvision.datasets as datasets

# Transforms
transform_train = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

transform_test = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

# Datasets (ImageFolder picks class names from subfolders)
trainset = datasets.ImageFolder(os.path.join(split_dir, "train"), transform=transform_train)
valset   = datasets.ImageFolder(os.path.join(split_dir, "val"), transform=transform_test)
testset  = datasets.ImageFolder(os.path.join(split_dir, "test"), transform=transform_test)

# Dataloaders
trainloader = DataLoader(trainset, batch_size=32, shuffle=True, num_workers=2)
valloader   = DataLoader(valset, batch_size=32, shuffle=False, num_workers=2)
testloader  = DataLoader(testset, batch_size=32, shuffle=False, num_workers=2)

print("Train size:", len(trainset))
print("Val size:", len(valset))
print("Test size:", len(testset))
print("Classes:", trainset.classes)

```

```
Train size: 3360
Val size: 420
Test size: 420
Classes: ['Normal', 'Tuberculosis']
```

```
# ============================
# 4. CvT Model Definition
# ============================
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from collections import Counter

# This is a dummy class to represent your custom CvT model.
# The user's provided code is copied here with the fix in the forward pass.
class CvT(nn.Module):
    def __init__(self, num_classes=2):
```

```
        super().__init__()
        # Simplified for demonstration. This is where your CvT layers go.
        # Your provided CvT model definition is integrated below.
        self.embed = ConvTokenEmbed(3, 64, kernel_size=7, stride=4)
        self.blocks = nn.Sequential(
            CvTBlock(64, num_heads=2),
            CvTBlock(64, num_heads=2),
        )
        self.norm = nn.LayerNorm(64)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, 64))
        self.fc = nn.Linear(64, num_classes)

    def forward(self, x):
        B = x.shape[0]
        x = self.embed(x)  # [B, N, 64] where N is number of image tokens

        # --- FIX: Correctly separating and re-concatenating tokens ---
        # Acknowledge and revert the previous bug introduced in the forward pass.
        # The class token (cls_tokens) and image tokens (x) must be handled separately
        # before passing through the attention blocks to avoid the shape error.
        # The ConvAttention module expects a perfect square number of tokens.

        cls_tokens = self.cls_token.expand(B, -1, -1)

        # Pass only the image tokens to the CvTBlocks, which use ConvAttention.
        img_tokens_out = self.blocks(x)

        # Now, concatenate the class token back with the processed image tokens.
        x = torch.cat((cls_tokens, img_tokens_out), dim=1)

        x = self.norm(x)

        # The final classification is based on the class token's output.
        return self.fc(x[:, 0])

class ConvTokenEmbed(nn.Module):
    def __init__(self, in_channels, embed_dim, kernel_size=7, stride=4, padding=3):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, embed_dim, kernel_size, stride, padding, bias=False)
        self.bn = nn.BatchNorm2d(embed_dim)
    def forward(self, x):
        x = F.relu(self.bn(self.conv(x)))
        return x.flatten(2).transpose(1, 2)

class ConvAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.scale = (embed_dim // num_heads) ** -0.5
        self.depthwise_conv = nn.Conv2d(embed_dim, embed_dim, kernel_size=3, padding=1, groups=embe
        self.pointwise_conv = nn.Conv2d(embed_dim, embed_dim * 3, kernel_size=1)
        self.proj = nn.Linear(embed_dim, embed_dim)
    def forward(self, x):
        B, N, C = x.shape
        x_reshaped = x.transpose(1, 2).view(B, C, int(N ** 0.5), int(N ** 0.5))
        qkv = self.pointwise_conv(self.depthwise_conv(x_reshaped)).view(B, 3 * C, N).transpose(1, 2
        qkv = qkv.reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = F.softmax(attn, dim=-1)
        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        return self.proj(x)

class CvTBlock(nn.Module):
    def __init__(self, embed_dim, num_heads):
```

```
        super().__init__()
        self.norm1 = nn.LayerNorm(embed_dim)
        self.attn = ConvAttention(embed_dim, num_heads)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, 4 * embed_dim),
            nn.GELU(),
            nn.Linear(4 * embed_dim, embed_dim)
        )
    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        x = x + self.mlp(self.norm2(x))
        return x


# ============================
# 5. Training Setup
# ============================
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CvT(num_classes=2).to(device)

# --- FIX: Weighted Loss Function to address data imbalance ---
# Assuming a 5:1 ratio of majority to minority class.
# We'll assign a weight of 5 to the minority class (index 1) and 1 to the majority (index 0).
# You should replace these values with the actual inverse class frequencies from your dataset.
weights = torch.tensor([1.0, 5.0]).to(device)
criterion = nn.CrossEntropyLoss(weight=weights)


optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.9, weight_decay=5e-3)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)

def train_one_epoch(model, loader, criterion, optimizer, device):
    model.train()
    running_loss, correct, total = 0.0, 0, 0
    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    return running_loss / len(loader), 100 * correct / total

def evaluate(model, loader, criterion, device):
    model.eval()
    running_loss, correct, total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return running_loss / len(loader), 100 * correct / total

```

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

```
import torch

# ============================
# 1. Class Distribution
# ============================
def plot_class_distribution(trainset, valset, testset):
    def get_labels(dataset):
        return [label for _, label in dataset]

    datasets_map = {
        "Train": get_labels(trainset),
        "Validation": get_labels(valset),
        "Test": get_labels(testset),
    }

    plt.figure(figsize=(15, 4))
    for i, (name, labels) in enumerate(datasets_map.items()):
        plt.subplot(1, 3, i + 1)
        sns.countplot(x=labels, palette="viridis")
        plt.title(f"{name} Set Distribution")
        plt.xticks(
            ticks=range(len(trainset.classes)),
            labels=trainset.classes,
            rotation=20
        )
    plt.tight_layout()
    plt.show()

plot_class_distribution(trainset, valset, testset)


# ============================
# 2. Sample Images
# ============================
def show_sample_images(dataset, class_names, n=5):
    plt.figure(figsize=(15, 3))
    for i in range(n):
        img, label = dataset[i]
        img = img.permute(1, 2, 0).numpy()
        img = np.clip(img * 0.229 + 0.485, 0, 1)  # unnormalize (approximate)
        plt.subplot(1, n, i + 1)
        plt.imshow(img)
        plt.title(class_names[label])
        plt.axis("off")
    plt.show()

show_sample_images(trainset, trainset.classes)


# ============================
# 3. Intensity Histograms
# ============================
def plot_intensity_histograms(dataset, n_samples=200):
    imgs = [dataset[i][0].permute(1, 2, 0).numpy() for i in range(min(n_samples, len(dataset)))]
    pixels = np.vstack([img.reshape(-1, 3) for img in imgs])

    plt.figure(figsize=(12, 4))
    colors = ["r", "g", "b"]
    for i, c in enumerate(colors):
        plt.hist(pixels[:, i], bins=50, color=c, alpha=0.5, label=f"{c}-channel")
    plt.title("Pixel Intensity Distribution (sample)")
    plt.legend()
    plt.show()

plot_intensity_histograms(trainset)

```

```
# ============================
# 4. Class Balance Heatmap
# ============================
def plot_class_heatmap(trainset, valset, testset):
    def count_labels(dataset):
        counts = torch.bincount(torch.tensor([y for _, y in dataset]))
        return counts.numpy()

    data = np.vstack([
        count_labels(trainset),
        count_labels(valset),
        count_labels(testset)
    ])

    sns.heatmap(data, annot=True, fmt="d", cmap="Blues",
                xticklabels=trainset.classes,
                yticklabels=["Train", "Val", "Test"])
    plt.title("Class Distribution Heatmap")
    plt.show()

plot_class_heatmap(trainset, valset, testset)


# ============================
# 5. Dataset Size Summary
# ============================
print(f"Train set size: {len(trainset)}")
print(f"Validation set size: {len(valset)}")
print(f"Test set size: {len(testset)}")
print(f"Classes: {trainset.classes}")

```

```
/tmp/ipython-input-4141679274.py:22: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set

  sns.countplot(x=labels, palette="viridis")
/tmp/ipython-input-4141679274.py:22: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set

  sns.countplot(x=labels, palette="viridis")
/tmp/ipython-input-4141679274.py:22: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set

  sns.countplot(x=labels, palette="viridis")
```

```python
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
import torch

# ============================
# 1. Training with logging
# ============================
num_epochs = 20
history = {
    "train_loss": [], "train_acc": [],
    "val_loss": [], "val_acc": [],
    "test_loss": [], "test_acc": []
}

for epoch in range(1, num_epochs + 1):
    train_loss, train_acc = train_one_epoch(model, trainloader, criterion, optimizer, device)
    val_loss, val_acc = evaluate(model, valloader, criterion, device)
    test_loss, test_acc = evaluate(model, testloader, criterion, device)

    scheduler.step()

    # log
    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)
    history["test_loss"].append(test_loss)
    history["test_acc"].append(test_acc)

    print(f"Epoch {epoch:02d}: "
          f"Train Acc {train_acc:.2f}% | "
          f"Val Acc {val_acc:.2f}% | "
          f"Test Acc {test_acc:.2f}%")


# ============================
# 2. Loss & Accuracy Curves
# ============================
def plot_training_curves(history):
    epochs = range(1, len(history["train_loss"]) + 1)

    plt.figure(figsize=(12,5))
    plt.subplot(1,2,1)
    plt.plot(epochs, history["train_loss"], label="Train")
    plt.plot(epochs, history["val_loss"], label="Val")
    plt.plot(epochs, history["test_loss"], label="Test")
    plt.title("Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1,2,2)
    plt.plot(epochs, history["train_acc"], label="Train")
```

```
    plt.plot(epochs, history["train_acc"], label="Train")
    plt.plot(epochs, history["val_acc"], label="Val")
    plt.plot(epochs, history["test_acc"], label="Test")
    plt.title("Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.legend()

    plt.show()

plot_training_curves(history)


# ============================
# 3. Confusion Matrix & Report
# ============================
def evaluate_predictions(model, loader, device, class_names):
    model.eval()
    all_preds, all_labels = [], []
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            preds = outputs.argmax(dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    return np.array(all_preds), np.array(all_labels)

# Use test set for evaluation
y_pred, y_true = evaluate_predictions(model, testloader, device, trainset.classes)

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=trainset.classes, yticklabels=trainset.classes)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()

# Classification Report
print("Classification Report:\n")
print(classification_report(y_true, y_pred, target_names=trainset.classes))


# ============================
# 4. ROC Curve (binary)
# ============================
if len(trainset.classes) == 2:
    from sklearn.preprocessing import label_binarize

    y_true_bin = label_binarize(y_true, classes=[0,1]).ravel()
    y_score = []
    model.eval()
    with torch.no_grad():
        for images, labels in testloader:
            images = images.to(device)
            outputs = model(images)
            probs = torch.softmax(outputs, dim=1)[:,1]  # probability of class 1
            y_score.extend(probs.cpu().numpy())
    y_score = np.array(y_score)

    fpr, tpr, _ = roc_curve(y_true_bin, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(6,5))
    plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
```

```
    plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
    plt.plot([0,1], [0,1], linestyle="--", color="gray")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend()
    plt.show()


# ============================
# 5. Misclassified Samples
# ============================
def show_misclassified(model, loader, class_names, device, n=6):
    model.eval()
    misclassified = []
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            preds = outputs.argmax(dim=1)
            for img, pred, true in zip(images, preds, labels):
                if pred != true:
                    misclassified.append((img.cpu(), pred.item(), true.item()))
                if len(misclassified) >= n:
                    break
            if len(misclassified) >= n:
                break

    plt.figure(figsize=(15, 5))
    for i, (img, pred, true) in enumerate(misclassified):
        plt.subplot(1, n, i+1)
        img = img.permute(1,2,0).numpy()
        img = np.clip(img*0.229 + 0.485, 0, 1)  # unnormalize approx
        plt.imshow(img)
        plt.title(f"P:{class_names[pred]}\nT:{class_names[true]}")
        plt.axis("off")
    plt.suptitle("Misclassified Samples", fontsize=16)
    plt.show()

show_misclassified(model, testloader, trainset.classes, device, n=6)
```