```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models
import matplotlib.pyplot as plt
import numpy as np
import time

# Device configuration (Use GPU if available)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

LEARNING_RATE=0.001
```

```
Using device: cuda
```

MOUNTING GOOGLE DRIVE

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

```
Mounted at /content/drive
```

```
!cp /content/drive/MyDrive/datasets/archive.zip /content/
```

UNZIPPING THE DATASET ZIP FILE

```
!unzip -q /content/archive.zip
```

```
import os
os.listdir("/content/kagglecatsanddogs_3367a/PetImages/")
```

```
['Dog', 'Cat']
```

CONFIRMING THE NUMBER OF IMAGES OF THESE TWO CLASSES

```
import os
print(os.listdir("/content/kagglecatsanddogs_3367a/PetImages/"))
print(len(os.listdir("/content/kagglecatsanddogs_3367a/PetImages/Cat")))
print(len(os.listdir("/content/kagglecatsanddogs_3367a/PetImages/Dog")))
```

```
['Dog', 'Cat']
12491
12470
```

SPLITTING INTO TRAINING AND VALIDATION DATASET

```
import os, shutil, random
from PIL import Image

SRC_DIR = "/content/kagglecatsanddogs_3367a/PetImages"
DST_DIR = "/content/cats_dogs"

SPLIT = 0.8
random.seed(42)

# Create folders
for split in ["train", "val"]:
    for cls in ["cat", "dog"]:
        os.makedirs(f"{DST_DIR}/{split}/{cls}", exist_ok=True)

def process_class(cls):
    src = os.path.join(SRC_DIR, cls.capitalize())
    files = os.listdir(src)
    random.shuffle(files)
    cut = int(len(files) * SPLIT)
```

```
    def copy(files, split):
        for f in files:
            src_file = os.path.join(src, f)
            dst_file = f"{DST_DIR}/{split}/{cls}/{f}"
            try:
                Image.open(src_file).verify()
                shutil.copy(src_file, dst_file)
            except:
                pass

    copy(files[:cut], "train")
    copy(files[cut:], "val")

process_class("cat")
process_class("dog")

print(" Train/Validation split created")
```

```
/usr/local/lib/python3.12/dist-packages/PIL/TiffImagePlugin.py:950: UserWarning: Truncated File Read
  warnings.warn(str(msg))
 Train/Validation split created
```

```
import os

print("Train cats:", len(os.listdir("/content/cats_dogs/train/cat")))
print("Train dogs:", len(os.listdir("/content/cats_dogs/train/dog")))
print("Val cats:", len(os.listdir("/content/cats_dogs/val/cat")))
print("Val dogs:", len(os.listdir("/content/cats_dogs/val/dog")))
```

```
Train cats: 9991
Train dogs: 9975
Val cats: 2499
Val dogs: 2494
```

SUPPRESSES PIL TRUNCATED WARNINGS

```
import warnings
from PIL import ImageFile

# Allow loading truncated images
ImageFile.LOAD_TRUNCATED_IMAGES = True

# Suppress PIL truncated image warnings
warnings.filterwarnings(
    "ignore",
    category=UserWarning,
    module="PIL"
)
```

DATASET LOADING

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

# ---------------- CONFIG ----------------
DATASET = "catsdogs"
BATCH_SIZE = 32

# ---------------- DEVICE ----------------
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)

# ---------------- TRANSFORMS ----------------
#for cats vs dog
IMG_SIZE = 128
NUM_CLASSES = 2

transform_train = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),
```

```
                          (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),
                          (0.5, 0.5, 0.5))
])

# ---------------- DATASET LOADING ----------------

print("Loading Cats vs Dogs...")
train_dataset = datasets.ImageFolder(
        root="/content/cats_dogs/train",
        transform=transform_train
    )
test_dataset = datasets.ImageFolder(
        root="/content/cats_dogs/val",
        transform=transform_test
    )


# ---------------- DATALOADERS ----------------
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=2,
    pin_memory=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=2,
    pin_memory=True
)

# ---------------- INFO ----------------
classes = train_dataset.classes
print("Classes:", classes)
print("Train samples:", len(train_dataset))
print("Test samples:", len(test_dataset))
```

```
Using device: cuda
Loading Cats vs Dogs...
Classes: ['cat', 'dog']
Train samples: 19966
Test samples: 4993
```

DEFINING THE CONFIGURABLE CNN ARCHITECTURE

```
import torch
import torch.nn as nn

class ConfigurableCNN(nn.Module):
    def __init__(self, activation_fn_name='relu', num_classes=2):
        super(ConfigurableCNN, self).__init__()

        # -------- Activation --------
        if activation_fn_name == 'relu':
            self.activation = nn.ReLU()
        elif activation_fn_name == 'tanh':
            self.activation = nn.Tanh()
        elif activation_fn_name == 'leaky_relu':
            self.activation = nn.LeakyReLU(0.01)

        # -------- Convolution Blocks --------
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
```

```
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

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

        # KEY FIX: adaptive pooling (input-size agnostic)
        self.adaptive_pool = nn.AdaptiveAvgPool2d((4, 4))

        # -------- Classifier --------
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        # Block 1
        x = self.pool(self.activation(self.bn1(self.conv1(x))))

        # Block 2
        x = self.pool(self.activation(self.bn2(self.conv2(x))))

        # Block 3
        x = self.pool(self.activation(self.bn3(self.conv3(x))))

        # Ensure fixed feature size
        x = self.adaptive_pool(x)

        # Safe flatten (preserves batch size)
        x = torch.flatten(x, start_dim=1)

        # Classifier
        x = self.fc1(x)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

INITIALIZATION TECHNIQUES

```
def apply_weight_init(model, init_type='random'):
    def init_weights(m):
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
            if init_type == 'xavier':
                nn.init.xavier_uniform_(m.weight)
            elif init_type == 'kaiming':
                nn.init.kaiming_uniform_(m.weight, nonlinearity='relu')
            elif init_type == 'random':
                nn.init.normal_(m.weight, mean=0.0, std=0.02) # Simple random normal

            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

    model.apply(init_weights)
```

TRAINING UTILITY FUNCTION

```
def train_model(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    history = {'loss': [], 'acc': []}

    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 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()
```

```
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

            epoch_acc = 100 * correct / total
            epoch_loss = running_loss / len(train_loader)
            history['loss'].append(epoch_loss)
            history['acc'].append(epoch_acc)

            print(f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, Acc: {epoch_acc:.2f}%")

        return history
```

EVALUATION UTILITY FUNCTION

```
def evaluate_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.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    acc = 100 * correct / total
    return acc
```

EXPERIMENT CONFIGURATIONS (AROUND 27 DIFFERENT CONFIG)

```
# Experiment Configurations
activations = ['relu', 'tanh', 'leaky_relu']
initializations = ['xavier', 'kaiming', 'random']
optimizers_list = ['sgd', 'adam', 'rmsprop']

results = []
best_acc = 0
best_config = {}
best_model_state = None

print("Starting Experiments...")

# Iterate through all combinations
for act in activations:
    for init in initializations:
        for opt_name in optimizers_list:
            print(f"\n--- Config: Act={act}, Init={init}, Optim={opt_name} ---")

            # 1. Initialize Model
            # model = ConfigurableCNN(activation_fn_name=act, num_classes=10).to(device)
            model = ConfigurableCNN(
            activation_fn_name=act,
            num_classes=NUM_CLASSES).to(device)
            # 2. Apply Weight Init
            apply_weight_init(model, init_type=init)

            # 3. Setup Optimizer
            criterion = nn.CrossEntropyLoss()
            if opt_name == 'sgd':
                optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE, momentum=0.9)
            elif opt_name == 'adam':
                optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
            elif opt_name == 'rmsprop':
                optimizer = optim.RMSprop(model.parameters(), lr=LEARNING_RATE)

            # 4. Train

            train_hist = train_model(model, train_loader, criterion, optimizer, epochs=10)

            # 5. Evaluate
            test_acc = evaluate_model(model, test_loader)
            print(f"Test Accuracy: {test_acc:.2f}%")
```

```
            # Store results
            results.append({
                'config': f"{act}_{init}_{opt_name}",
                'acc': test_acc
            })

            # Check for best model
            if test_acc > best_acc:
                best_acc = test_acc
                best_config = {'act': act, 'init': init, 'opt': opt_name}
                best_model_state = model.state_dict()
                torch.save(model.state_dict(), "best_custom_cnn.pth")

print(f"\nBest Custom Configuration: {best_config} with Acc: {best_acc:.2f}%")
```

```
Epoch [8/10], Loss: 0.3081, Acc: 86.84%
Epoch [9/10], Loss: 0.2902, Acc: 87.79%
Epoch [10/10], Loss: 0.2715, Acc: 88.68%
Test Accuracy: 88.66%

--- Config: Act=leaky_relu, Init=kaiming, Optim=rmsprop ---
Epoch [1/10], Loss: 0.8630, Acc: 64.20%
Epoch [2/10], Loss: 0.5424, Acc: 73.25%
Epoch [3/10], Loss: 0.4801, Acc: 77.30%
Epoch [4/10], Loss: 0.4298, Acc: 80.60%
Epoch [5/10], Loss: 0.3956, Acc: 82.34%
Epoch [6/10], Loss: 0.3652, Acc: 83.79%
Epoch [7/10], Loss: 0.3417, Acc: 85.37%
Epoch [8/10], Loss: 0.3158, Acc: 86.25%
Epoch [9/10], Loss: 0.2949, Acc: 87.57%
Epoch [10/10], Loss: 0.2747, Acc: 88.18%
Test Accuracy: 84.02%

--- Config: Act=leaky_relu, Init=random, Optim=sgd ---
Epoch [1/10], Loss: 0.5962, Acc: 67.68%
Epoch [2/10], Loss: 0.5140, Acc: 74.80%
Epoch [3/10], Loss: 0.4659, Acc: 78.06%
Epoch [4/10], Loss: 0.4322, Acc: 79.97%
Epoch [5/10], Loss: 0.4141, Acc: 81.12%
Epoch [6/10], Loss: 0.3955, Acc: 82.42%
Epoch [7/10], Loss: 0.3762, Acc: 83.10%
Epoch [8/10], Loss: 0.3685, Acc: 83.64%
Epoch [9/10], Loss: 0.3608, Acc: 84.19%
Epoch [10/10], Loss: 0.3474, Acc: 84.77%
Test Accuracy: 82.38%

--- Config: Act=leaky_relu, Init=random, Optim=adam ---
Epoch [1/10], Loss: 0.6227, Acc: 65.59%
Epoch [2/10], Loss: 0.5432, Acc: 72.45%
Epoch [3/10], Loss: 0.4992, Acc: 75.75%
Epoch [4/10], Loss: 0.4575, Acc: 78.50%
Epoch [5/10], Loss: 0.4254, Acc: 80.77%
Epoch [6/10], Loss: 0.4046, Acc: 81.73%
Epoch [7/10], Loss: 0.3741, Acc: 83.64%
Epoch [8/10], Loss: 0.3483, Acc: 84.74%
Epoch [9/10], Loss: 0.3321, Acc: 85.53%
Epoch [10/10], Loss: 0.3071, Acc: 86.65%
Test Accuracy: 79.39%

--- Config: Act=leaky_relu, Init=random, Optim=rmsprop ---
Epoch [1/10], Loss: 0.7596, Acc: 62.56%
Epoch [2/10], Loss: 0.5635, Acc: 70.99%
Epoch [3/10], Loss: 0.5133, Acc: 74.79%
Epoch [4/10], Loss: 0.4614, Acc: 78.73%
Epoch [5/10], Loss: 0.4244, Acc: 80.80%
Epoch [6/10], Loss: 0.3897, Acc: 82.68%
Epoch [7/10], Loss: 0.3590, Acc: 84.22%
Epoch [8/10], Loss: 0.3334, Acc: 85.28%
Epoch [9/10], Loss: 0.3101, Acc: 86.62%
Epoch [10/10], Loss: 0.2890, Acc: 87.41%
Test Accuracy: 79.83%

Best Custom Configuration: {'act': 'leaky_relu', 'init': 'kaiming', 'opt': 'adam'} with Acc: 88.66%
```

```
# RESNET-18 TRANSFER LEARNING
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader

#          DEVICE
```

```
# -------- DEVICE --------
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)

# -------- TRANSFORMS (ImageNet) --------
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]
    )
])

# -------- DATASET --------
train_dataset = datasets.ImageFolder(
    root="/content/cats_dogs/train",
    transform=transform_train
)

test_dataset = datasets.ImageFolder(
    root="/content/cats_dogs/val",
    transform=transform_test
)

train_loader = DataLoader(
    train_dataset, batch_size=32,
    shuffle=True, num_workers=2, pin_memory=True
)

test_loader = DataLoader(
    test_dataset, batch_size=32,
    shuffle=False, num_workers=2, pin_memory=True
)

print("Classes:", train_dataset.classes)

# -------- MODEL --------
print("\n--- Starting Transfer Learning (ResNet-18 | Cats vs Dogs) ---")

resnet = models.resnet18(
    weights=models.ResNet18_Weights.IMAGENET1K_V1
)

# Freeze backbone
for param in resnet.parameters():
    param.requires_grad = False

# Replace FC layer
num_ftrs = resnet.fc.in_features
resnet.fc = nn.Linear(num_ftrs, 2)

resnet = resnet.to(device)

# -------- TRAINING --------
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet.fc.parameters(), lr=0.001)

train_model(
    resnet,
    train_loader,
    criterion,
    optimizer,
    epochs=20
)

# -------- EVALUATION --------
resnet_acc = evaluate_model(resnet, test_loader)
```

```
print(f"\nResNet-18 Accuracy (Cats vs Dogs): {resnet_acc:.2f}%")
```

```
Using device: cuda
Classes: ['cat', 'dog']

--- Starting Transfer Learning (ResNet-18 | Cats vs Dogs) ---
Epoch [1/20], Loss: 0.1219, Acc: 95.37%
Epoch [2/20], Loss: 0.0860, Acc: 96.70%
Epoch [3/20], Loss: 0.0769, Acc: 96.95%
Epoch [4/20], Loss: 0.0756, Acc: 97.04%
Epoch [5/20], Loss: 0.0717, Acc: 97.26%
Epoch [6/20], Loss: 0.0757, Acc: 97.00%
Epoch [7/20], Loss: 0.0745, Acc: 97.18%
Epoch [8/20], Loss: 0.0688, Acc: 97.39%
Epoch [9/20], Loss: 0.0670, Acc: 97.40%
Epoch [10/20], Loss: 0.0697, Acc: 97.30%
Epoch [11/20], Loss: 0.0700, Acc: 97.23%
Epoch [12/20], Loss: 0.0736, Acc: 97.17%
Epoch [13/20], Loss: 0.0673, Acc: 97.34%
Epoch [14/20], Loss: 0.0686, Acc: 97.32%
Epoch [15/20], Loss: 0.0631, Acc: 97.59%
Epoch [16/20], Loss: 0.0694, Acc: 97.43%
Epoch [17/20], Loss: 0.0697, Acc: 97.35%
Epoch [18/20], Loss: 0.0722, Acc: 97.18%
Epoch [19/20], Loss: 0.0654, Acc: 97.54%
Epoch [20/20], Loss: 0.0710, Acc: 97.26%

ResNet-18 Accuracy (Cats vs Dogs): 98.10%
```