```
# Step 1: Change runtime type from cpu to gpu (for parallel execution
to speed up) in colab
import torch

torch.cuda.is_available()

True

if torch.cuda.is_available():
  print(f"Assigned GPU is : {torch.cuda.get_device_name(0)}")
  !nvidia-smi

Assigned GPU is : Tesla T4
Fri Aug 29 01:39:42 2025
+----------------------------------------------------------------------
--------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15
CUDA Version: 12.4     |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A |
Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage |
GPU-Util  Compute M. |
|                                         |                        |
MIG M. |
|
=========================================+========================+===
===================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |
0 |
| N/A   44C    P8             11W /   70W |       2MiB /  15360MiB |
0%      Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------
+----------------------+


+----------------------------------------------------------------------
--------------------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
|        ID   ID
Usage      |
|
======================================================================
===================|
|  No running processes found
|
+-----------------------------------------------------------------------
--------------------+
```

```python
print(f"Python version: {platform.python_version()}")
```

```
Python version: 3.12.11
```

```python
print(f"PyTorch version: {torch.__version__}")
```

```
PyTorch version: 2.8.0+cu126
```

```python
# Runtime → Change runtime type → Hardware accelerator: GPU (then run)
import torch, platform, subprocess, sys
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("GPU:", torch.cuda.get_device_name(0))
    !nvidia-smi
print("Python:", platform.python_version())
print("PyTorch:", torch.__version__)
```

```
CUDA available: True
GPU: Tesla T4
Fri Aug 29 01:39:42 2025
+-----------------------------------------------------------------------
--------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15
CUDA Version: 12.4     |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A |
Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage |
GPU-Util  Compute M. |
|                                         |                        |
MIG M. |
|
=========================================+========================+===
===================|
|   0  Tesla T4                      Off |   00000000:00:04.0 Off |
0 |
| N/A   44C    P8             10W /   70W |       2MiB /  15360MiB |
0%      Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------
+----------------------+


+-----------------------------------------------------------------------
```

```
--------------------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
|        ID   ID
Usage      |
|
======================================================================
===================|
|  No running processes found
|
+----------------------------------------------------------------------
--------------------+
Python: 3.12.11
PyTorch: 2.8.0+cu126
```

```
# Step 2: Set random seeds to make training more stable -
Reproducability
import random
import numpy as np

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

if torch.cuda.is_available():
  torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"{device}")
```

```
cuda
```

```
# Step 3: Configure PyTorch to try & choose the fastest, non-
deterministic algorithm

# Let PyTorch (internally via cudnn) - try different algorithms &
choose the fastest one
torch.backends.cudnn.benchmark = True

# Let PyTorch get faster results by using non-deterministic algorithms
torch.backends.cudnn.deterministic = False

# Step 4: Load CIFAR-10 image dataset - data pre-processing &
augmentation

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

```
# Define transformations to be applied on image dataset

# CIFAR-10 mean/std values per channel - RGB
MEAN = (0.4914, 0.4822, 0.4465)
STD = (0.2470, 0.2435, 0.2616)

# Create data augmentation (to improve robustness) & normalization on
training data
train_tfms = T.Compose([T.RandomCrop(32, padding=4),
T.RandomHorizontalFlip(), T.ToTensor(), T.Normalize(MEAN, STD)])

# Create data normalization on test data
test_tfms  = T.Compose([T.ToTensor(), T.Normalize(MEAN, STD)])

# Step 5: Download datasets

root_folder = "./data"   # folder where image data will be stored

# 1. Get CIFAR10 Train dataset
# Create transforms - crop + flip (to improve robustness) + convert to
Tensor + normalize
train_tfms = T.Compose([T.RandomCrop(32, padding=4),
T.RandomHorizontalFlip(), T.ToTensor(), T.Normalize(MEAN, STD)])

# Get training data after applying transforms
train_ds = torchvision.datasets.CIFAR10(root_folder, train=True,
download=True, transform=train_tfms)

# 2. Get CIFAR10 Test dataset
# Create transforms - convert to Tensor + normalize
test_tfms  = T.Compose([T.ToTensor(), T.Normalize(MEAN, STD)])

# Get test data after applying transforms
test_ds  = torchvision.datasets.CIFAR10(root_folder, train=False,
download=True, transform=test_tfms)
```

```
100%|██████████| 170M/170M [00:03<00:00, 44.8MB/s]
```

```
classes = train_ds.classes

BATCH_SIZE = 128
NUM_WORKERS = 2
PIN_MEMORY = True if torch.cuda.is_available() else False

train_loader = DataLoader(train_ds,
                          batch_size=BATCH_SIZE,
                          shuffle=True,
                          num_workers=NUM_WORKERS,
                          pin_memory=PIN_MEMORY,
                          persistent_workers=True)
```

```
test_loader  = DataLoader(test_ds,
                          batch_size=BATCH_SIZE,
                          shuffle=False,
                          num_workers=NUM_WORKERS,
                          pin_memory=PIN_MEMORY,
                          persistent_workers=True)

# Peek at a batch - for sanity checking the shape of images
xb, yb = next(iter(train_loader))
print(f"Batch images: {xb.shape}, Batch labels: {yb.shape}, Label
sample: {yb[:10].tolist()}")
```

```
Batch images: torch.Size([128, 3, 32, 32]), Batch labels:
torch.Size([128]), Label sample: [6, 0, 4, 1, 2, 7, 9, 4, 7, 8]
```

```
# Build a custom CNN

# stacks of Conv -> BatchNorm -> ReLU -> MaxPool - to extract
increasingly abstract features
# add dropout to regularize

import torch.nn.functional as F

class SimpleCNN(torch.nn.Module):
  def __init__(self, num_classes=10, p_drop=0.5):
     super().__init__()

     # Block 1: 3 -> 32
     self.conv1 = torch.nn.Conv2d(in_channels=3, out_channels=32,
kernel_size=3, padding=1)
     self.bn1 = torch.nn.BatchNorm2d(32)

     # Block 2: 32 -> 64
     self.conv2 = torch.nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, padding=1)
     self.bn2 = torch.nn.BatchNorm2d(64)

     # Block 3: 64 -> 128
     self.conv3 = torch.nn.Conv2d(in_channels=64, out_channels=128,
kernel_size=3, padding=1)
     self.bn3 = torch.nn.BatchNorm2d(128)

     # Block 4: 128 -> 256
     self.conv4 = torch.nn.Conv2d(in_channels=128, out_channels=256,
kernel_size=3, padding=1)
     self.bn4 = torch.nn.BatchNorm2d(256)

     # Block 5: 256 -> 512
     self.conv5 = torch.nn.Conv2d(in_channels=256, out_channels=512,
kernel_size=3, padding=1)
```

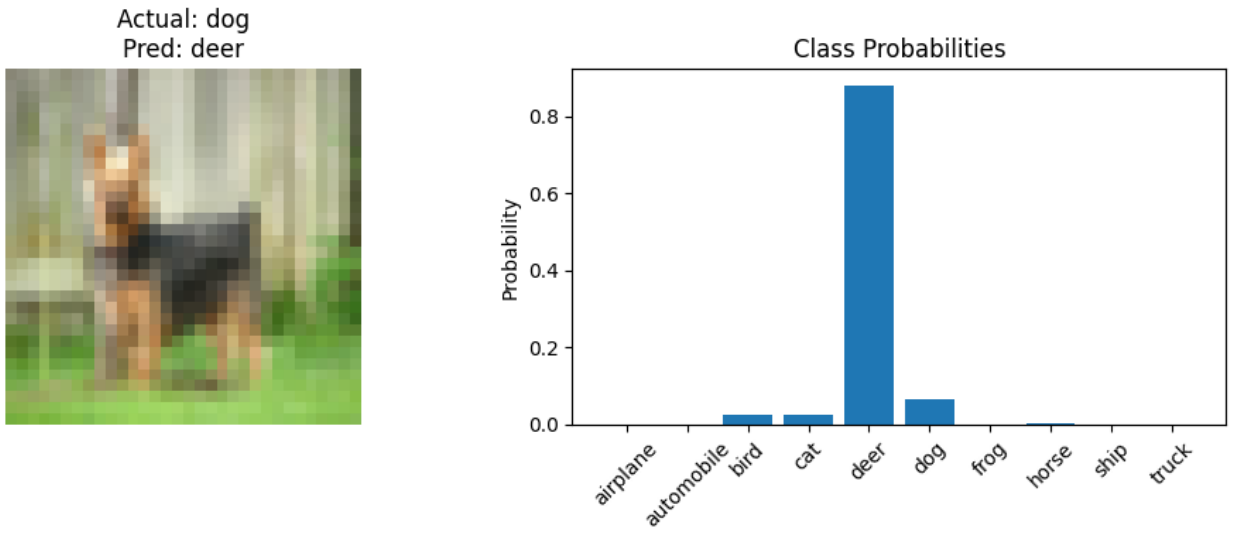
```
      self.bn5 = torch.nn.BatchNorm2d(512)

      self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2)
      self.drop = torch.nn.Dropout(p=p_drop)

      # After 4 pools on 32 x 32 -> 16 x 16 -> 8 x 8 -> 4 x 4 -> 2 x 2
-> 1 x 1
      self.fc1 = torch.nn.Linear(in_features=1*1*512, out_features=256)
      self.fc2 = torch.nn.Linear(in_features=256, out_features=10)

   def forward(self, x):
     x = self.pool(F.relu(self.bn1(self.conv1(x))))  # out: 32, 16, 16
     x = self.pool(F.relu(self.bn2(self.conv2(x))))  # out: 64, 8, 8
     x = self.pool(F.relu(self.bn3(self.conv3(x))))  # out: 128, 4, 4
     x = self.pool(F.relu(self.bn4(self.conv4(x))))  # out: 256, 2, 2
     x = self.pool(F.relu(self.bn5(self.conv5(x))))  # out: 512, 1, 1
     x = torch.flatten(x, start_dim=1)               # out: 256 * 2 * 2
     x = self.drop(F.relu(self.fc1(x)))              # out: after
dropout 50% (to regularize/ generalize)
     x = self.fc2(x)                                 # out: final
predicted class (out of 10 classes)
     return x

model = SimpleCNN().to(device)
print(model)

SimpleCNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv5): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (bn5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (drop): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=512, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)

# Set training parameters - loss fn, learning rate - optimizer &
scheduler
# Use AMP for time & space efficiency - when using gpu (parallel
computation), uses even faster computation & consumes lesser memory)

loss_fn = torch.nn.CrossEntropyLoss()

base_lr = 1e-3
lr_optimizer = torch.optim.AdamW(model.parameters(), lr=base_lr,
weight_decay=1e-4)

#lr_scheduler = torch.optim.lr_scheduler.StepLR(lr_optimizer,
step_size=10, gamma=0.5) # half LR every 10 epochs
num_epochs = 10
lr_scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(lr_optimizer,
T_max=num_epochs, eta_min=1e-6)

# Turn AMP (Automatic Mixed Precision) only if cuda GPU is available
# Setting USE_AMP to False to avoid GradScaler error
#USE_AMP = False # and torch.cuda.is_available()
USE_AMP = True

# AMP uses FP16 (floating point 16) instead of FP32 default in GPU,
which speeds up computation
# But AMP (FP16) can shrink gradient values to zero, so we use
GradScaler to scale up the gradients
scaler = torch.amp.GradScaler('cuda', enabled=USE_AMP)

# Keeping the training loop clean - evaluate with no gradients

def accuracy_from_logits(logits, y):
  return (logits.argmax(1) == y).float().mean().item()

@torch.no_grad()
def evaluate(model, loader):
  model.eval()

  total_loss = 0.0
  total_accuracy = 0.0
  total_samples = 0

  for xb, yb in loader: # Iterate over batches of data (images & their
labels)
    # xb = images in batch, yb = labels in batch

    # Move images & labels to device (GPU for faster computation)
    xb = xb.to(device, non_blocking=True)
    yb = yb.to(device, non_blocking=True)

    if USE_AMP:
      with torch.cuda.amp.autocast():
        #print(f"Using CUDA Autocast...")
        logits = model(xb)
        loss = loss_fn(logits, yb)
    else:
      logits = model(xb)
      loss = loss_fn(logits, yb)

    batch_size = xb.size(0)
    total_loss = total_loss + loss.item() * batch_size
    total_accuracy = total_accuracy + accuracy_from_logits(logits, yb)
* batch_size # Removed .item() here
    total_samples = total_samples + batch_size

  return total_loss / total_samples, total_accuracy / total_samples

# Train the model - control epochs, metrics, save best weights
import time

EPOCHS = 10
best_state = None
best_accuracy = 0.0
t0_all = time.time()

# Move model to the device before starting training
model.to(device)

for epoch in range(EPOCHS):
  model.train()
  t0 = time.time()

  running_loss = 0.0
  running_accuracy = 0.0
  n = 0

  for xb, yb in train_loader:
    # Get each batch of images & their labels
    xb = xb.to(device, non_blocking=True)
    yb = yb.to(device, non_blocking=True)

    lr_optimizer.zero_grad(set_to_none=True)

    # Calculate logits and loss
    if USE_AMP:
        with torch.cuda.amp.autocast():
            logits = model(xb)
            loss = loss_fn(logits, yb)
    else:
        logits = model(xb)
        loss = loss_fn(logits, yb)

    if USE_AMP:
      # AMP-specific backward and step
      scaler.scale(loss).backward()
      scaler.step(lr_optimizer)
      scaler.update()
      #print(f"Using AMP...")
    else:
      # Standard backward and step
      loss.backward()
      lr_optimizer.step()

    batch_size = xb.size(0)
    running_loss = running_loss + loss.item() * batch_size
    running_accuracy = running_accuracy + accuracy_from_logits(logits,
yb) * batch_size # Removed .item() here
    n = n + batch_size

    #print(f"batch size = {batch_size}, total samples seen = {n}")
    #print(f"running accuracy = {running_accuracy}")

  lr_scheduler.step()
  train_loss, train_acc = running_loss / n, running_accuracy / n
  test_loss, test_acc = evaluate(model, test_loader)

  if test_acc > best_accuracy:
    best_accuracy = test_acc
    best_state = {k: v.detach().cpu() for k, v in
model.state_dict().items()}

  print(f"Epoch {epoch:02d}/{EPOCHS}"
        f"| train accuracy {train_acc*100: 5.2f}% loss
{train_loss:.4f} "
        f"| test accuracy {test_acc*100: 5.2f}% loss {test_loss:.4f} "
        f"| learning_rate {lr_scheduler.get_last_lr()[0]:.5f} "
        f"| {time.time()-t0:.1f}s")

  print(f"CUDA available = {torch.cuda.is_available()}")
  if torch.cuda.is_available():
    print(f"CUDA / GPU device name = {torch.cuda.get_device_name(0)}")

"""
  # Model and batch devices
  print(f"Model device = {next(model.parameters()).device}")
  print(f"Batch device = {xb.device}")
  #!nvidia-smi
  print(f"AMP flag = {USE_AMP}")
  print(f"GradScaler enabled: {isinstance(scaler,
torch.cuda.amp.GradScaler) and scaler.is_enabled()}")
  # cuDNN autotune status
  print(f"cudnn.benchmark = {torch.backends.cudnn.benchmark}")
  # Memory snapshot before & after a batch (Confirms GPU allocation
climb)
  print(f"Allocated MB: {torch.cuda.memory_allocated()/1e6}")
  print(f"Reserved MB: {torch.cuda.memory_reserved()/1e6}")
"""

print("Best test acc:", round(best_accuracy*100, 2), "%", "| total
time:", round(time.time()-t0_all, 1), "s")

# restore & save best
if best_state is not None:
    model.load_state_dict(best_state)
torch.save(model.state_dict(), "simplecnn_cifar10_best.pth")
```

```
/tmp/ipython-input-564692797.py:29: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.cuda.amp.autocast():
/tmp/ipython-input-1271363474.py:22: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.cuda.amp.autocast():

Epoch 00/10| train accuracy  45.05% loss 1.4859 | test accuracy
45.85% loss 1.5765 | learning_rate 0.00098 | 38.5s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 01/10| train accuracy  60.97% loss 1.1000 | test accuracy
67.13% loss 0.9207 | learning_rate 0.00090 | 21.3s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 02/10| train accuracy  67.17% loss 0.9367 | test accuracy
68.89% loss 0.8658 | learning_rate 0.00079 | 20.4s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 03/10| train accuracy  71.42% loss 0.8250 | test accuracy
72.27% loss 0.7961 | learning_rate 0.00065 | 19.2s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 04/10| train accuracy  74.69% loss 0.7388 | test accuracy
76.53% loss 0.6714 | learning_rate 0.00050 | 20.3s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 05/10| train accuracy  77.16% loss 0.6685 | test accuracy
76.09% loss 0.6773 | learning_rate 0.00035 | 19.1s
```

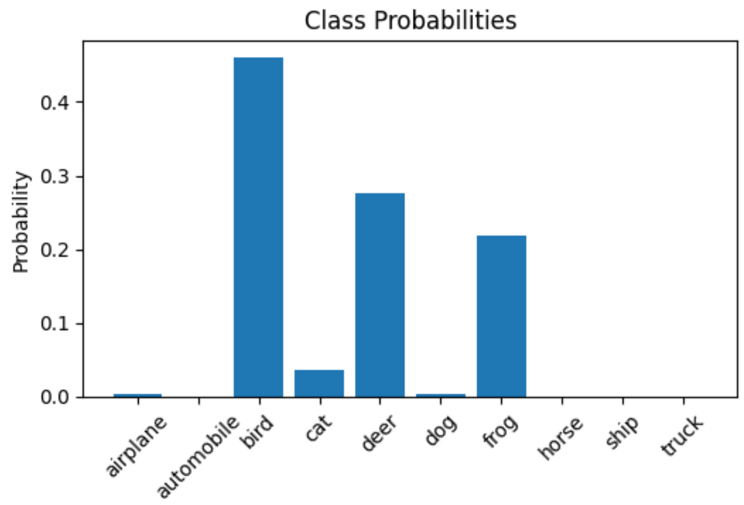
```
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 06/10| train accuracy  79.12% loss 0.6075 | test accuracy
79.82% loss 0.5880 | learning_rate 0.00021 | 19.9s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 07/10| train accuracy  80.98% loss 0.5597 | test accuracy
81.06% loss 0.5484 | learning_rate 0.00010 | 19.0s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 08/10| train accuracy  82.28% loss 0.5216 | test accuracy
81.92% loss 0.5218 | learning_rate 0.00003 | 19.9s
CUDA available = True
CUDA / GPU device name = Tesla T4
Epoch 09/10| train accuracy  83.00% loss 0.5029 | test accuracy
82.46% loss 0.5119 | learning_rate 0.00000 | 19.2s
CUDA available = True
CUDA / GPU device name = Tesla T4
Best test acc: 82.46 % | total time: 216.8 s

# Predict image class for a single instance to test

# Pick one sample from the test dataset
idx = 902
img, label = test_ds[idx]   # idx = sample we want to test
print(f"Actual label: {classes[label]}")

# Run it through model
model.eval()
with torch.no_grad():
  x = img.unsqueeze(0).to(device)
  logits = model(x)
  probs = torch.nn.functional.softmax(logits, dim=1)
  pred = probs.argmax(1).item()

print(f"Predicted label: {classes[pred]}\n")

for i, cls in enumerate(classes):
  print(f"{cls}: {probs[0][i].item()*100:.2f}%")

#print(f"Probabilites: {probs.cpu().numpy()}")

# Visualize the image
import matplotlib.pyplot as plt

plt.imshow(img.permute(1,2,0)*torch.tensor(STD) + torch.tensor(MEAN))
#un-normalize
plt.title(f"Actual: {classes[label]} | Predicted: {classes[pred]}")
plt.axis("off")
plt.show()
```

```
Actual label: frog
Predicted label: frog

airplane: 0.00%
automobile: 0.00%
bird: 0.00%
cat: 0.00%
deer: 0.00%
dog: 0.00%
frog: 100.00%
horse: 0.00%
ship: 0.00%
truck: 0.00%
```

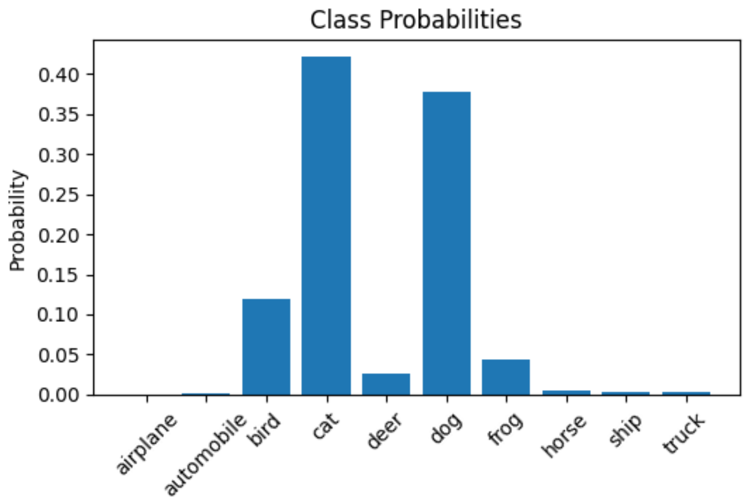
```python
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F

def show_probs(img, label, model, classes):
    model.eval()
    with torch.no_grad():
        x = img.unsqueeze(0).to(device)
        logits = model(x)
        probs = F.softmax(logits, dim=1).squeeze(0).cpu().numpy()
        pred = np.argmax(probs)
    if pred != label:  # only show misclassifications
        plt.figure(figsize=(10,4))

        # Image (left)
        plt.subplot(1,2,1)
        m = torch.tensor(MEAN).view(3,1,1)
        s = torch.tensor(STD).view(3,1,1)
        unnorm = (img * s + m).clamp(0,1)
        plt.imshow(unnorm.permute(1,2,0))
        plt.title(f"Actual: {classes[label]}\nPred: {classes[pred]}")
        plt.axis("off")

        # Probabilities (right)
        plt.subplot(1,2,2)
        plt.bar(range(len(classes)), probs)
        plt.xticks(range(len(classes)), classes, rotation=45)
        plt.ylabel("Probability")
        plt.title("Class Probabilities")
        plt.tight_layout()
        plt.show()

# □ Loop over some test samples
for i in range(50):   # check first 50 images
    img, label = test_ds[i]
    show_probs(img, label, model, classes)
```



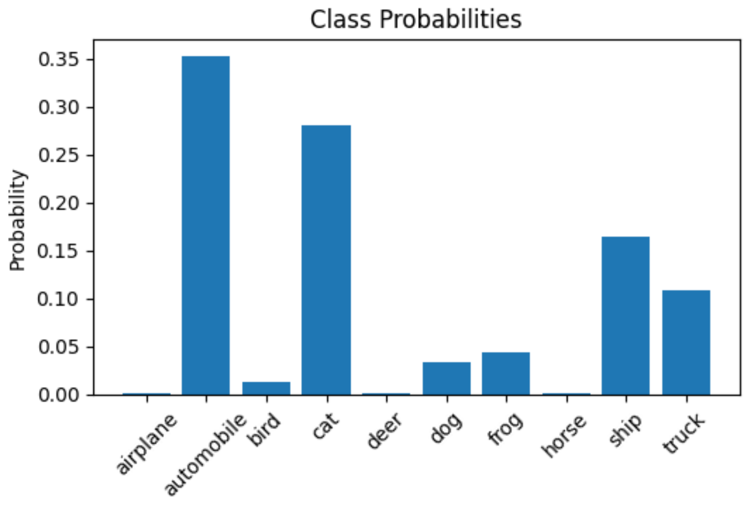
Actual: deer
Pred: bird

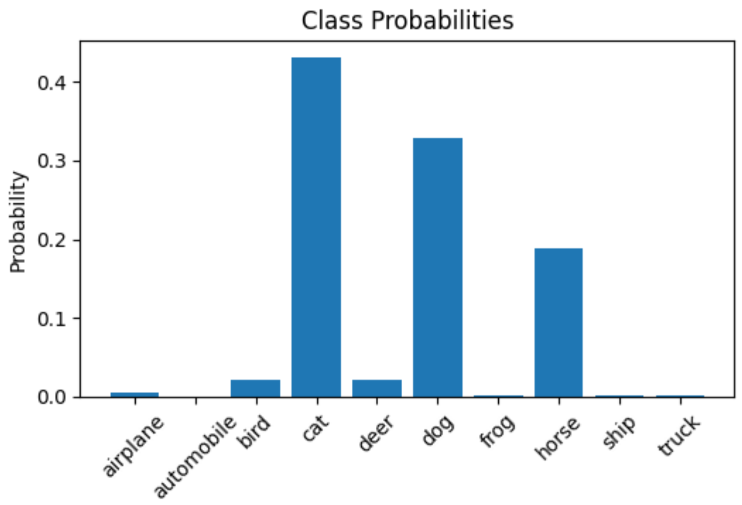

Class Probabilities
Probability
0.0
0.1
0.2
0.3
0.4
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Actual: dog
Pred: cat


Class Probabilities
Probability
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Actual: bird
Pred: automobile

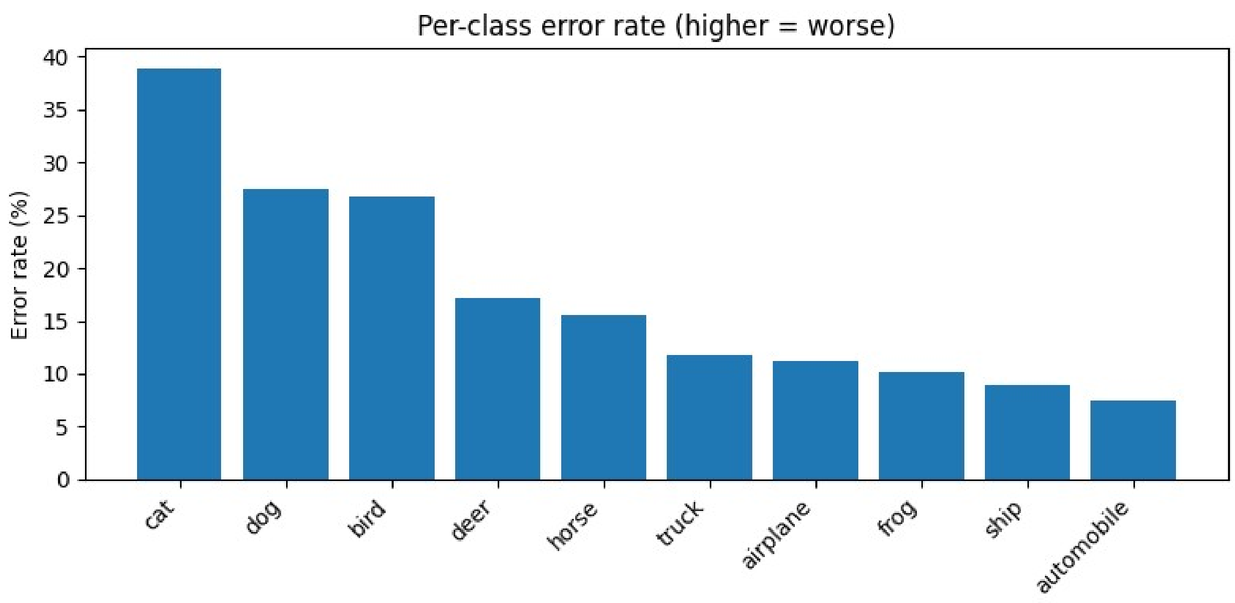

Class Probabilities
Probability
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck



```
# Collect predictions over the entire test dataset

import torch
import torch.nn.functional as F
import numpy as np
from sklearn.metrics import confusion_matrix, classification_report

model.eval()
all_preds, all_labels = [], []

with torch.no_grad():
    for xb, yb in test_loader:
        xb = xb.to(device, non_blocking=True)
        logits = model(xb)
        preds = logits.argmax(1).cpu()
        all_preds.append(preds)
        all_labels.append(yb)

y_true = torch.cat(all_labels).numpy()
y_pred = torch.cat(all_preds).numpy()

print(classification_report(y_true, y_pred, target_names=classes))
cm = confusion_matrix(y_true, y_pred)  # shape [num_classes,
num_classes]
```

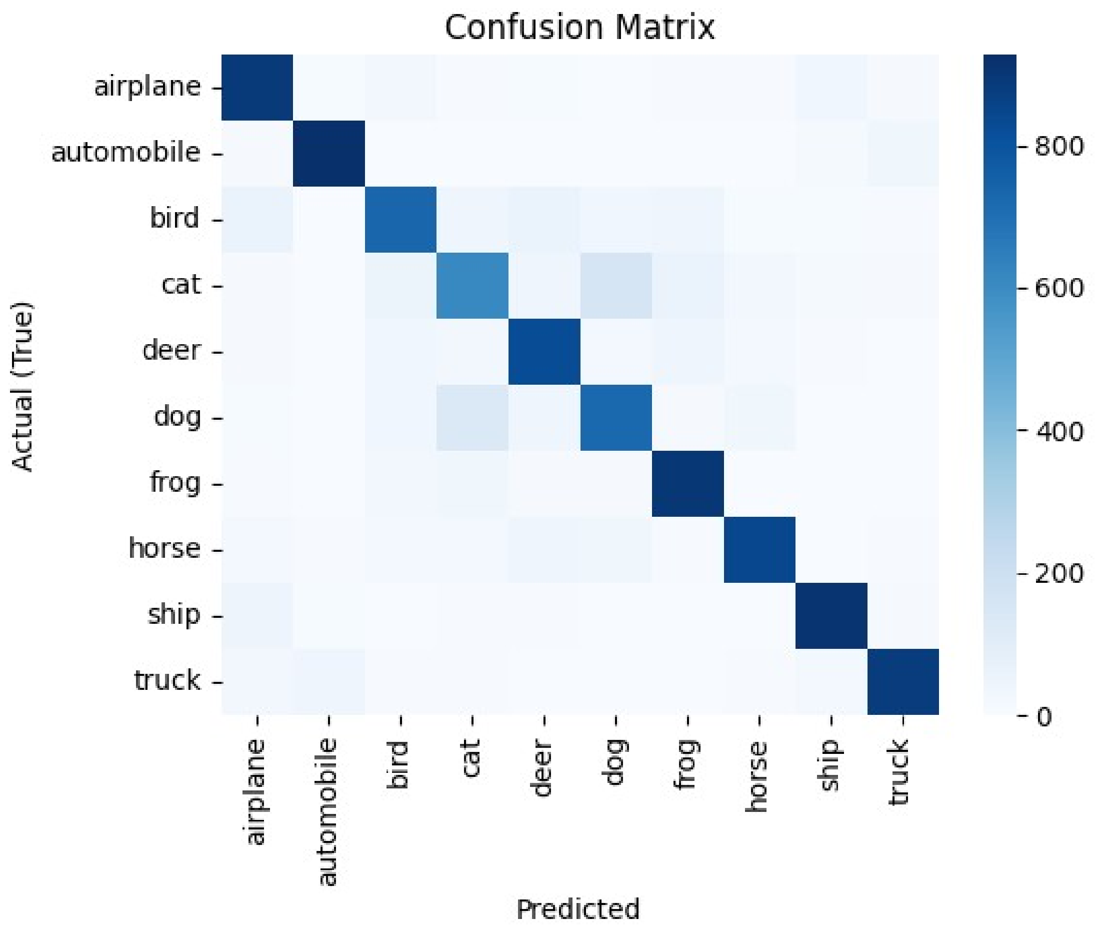
```
              precision    recall  f1-score   support

    airplane       0.81      0.89      0.85      1000
  automobile       0.93      0.93      0.93      1000
        bird       0.78      0.73      0.76      1000
         cat       0.68      0.61      0.65      1000
        deer       0.79      0.83      0.81      1000
         dog       0.73      0.72      0.73      1000
        frog       0.84      0.90      0.87      1000
       horse       0.88      0.84      0.86      1000
        ship       0.89      0.91      0.90      1000
       truck       0.91      0.88      0.89      1000

    accuracy                           0.82     10000
   macro avg       0.82      0.82      0.82     10000
weighted avg       0.82      0.82      0.82     10000
```

```python
# Compute per class error rates and plot

import matplotlib.pyplot as plt
import numpy as np

# per-class totals and errors (by true class)
per_class_total = cm.sum(axis=1)                     # row sums
per_class_correct = np.diag(cm)                      # diagonal
per_class_errors = per_class_total - per_class_correct
per_class_error_rate = 100 * per_class_errors / per_class_total

# sort by highest error rate
order = np.argsort(-per_class_error_rate)
sorted_classes = [classes[i] for i in order]
sorted_err = per_class_error_rate[order]

plt.figure(figsize=(8,4))
plt.bar(range(len(sorted_classes)), sorted_err)
plt.xticks(range(len(sorted_classes)), sorted_classes, rotation=45,
ha='right')
plt.ylabel("Error rate (%)")
plt.title("Per-class error rate (higher = worse)")
plt.tight_layout()
plt.show()

print("Worst classes (highest error %):")
for i in range(len(classes)):
    print(f"{sorted_classes[i]:10s} -> {sorted_err[i]:5.2f}%")
```



```
Worst classes (highest error %):
cat        -> 38.80%
dog        -> 27.50%
bird       -> 26.70%
deer       -> 17.20%
horse      -> 15.50%
truck      -> 11.80%
airplane   -> 11.20%
frog       -> 10.10%
ship       ->  8.90%
automobile ->  7.40%
```

```
# Confusion matrix heatmap

import seaborn as sns  # if not installed in Colab: !pip -q install
seaborn
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=False, fmt="d", cmap="Blues",
            xticklabels=classes, yticklabels=classes)
plt.xlabel("Predicted")
plt.ylabel("Actual (True)")
plt.title("Confusion Matrix")
plt.tight_layout()
plt.show()
```



```python
# Show "most confused" pairs

# For each true class, show the top wrong predicted class
for i, cls in enumerate(classes):
    row = cm[i].copy()
    row[i] = 0  # zero-out the diagonal
    j = row.argmax()
    if row.sum() > 0:
        print(f"True '{cls}' most confused with '{classes[j]}' "
              f"({row[j]} times, {100*row[j]/per_class_total[i]:.2f}%
of its errors)")
```

```
True 'airplane' most confused with 'ship' (36 times, 3.60% of its
errors)
True 'automobile' most confused with 'truck' (38 times, 3.80% of its
errors)
True 'bird' most confused with 'deer' (65 times, 6.50% of its errors)
True 'cat' most confused with 'dog' (160 times, 16.00% of its errors)
True 'deer' most confused with 'frog' (40 times, 4.00% of its errors)
True 'dog' most confused with 'cat' (133 times, 13.30% of its errors)
True 'frog' most confused with 'cat' (38 times, 3.80% of its errors)
True 'horse' most confused with 'deer' (41 times, 4.10% of its errors)
True 'ship' most confused with 'airplane' (50 times, 5.00% of its
errors)
True 'truck' most confused with 'automobile' (46 times, 4.60% of its
errors)
```

Now trying with pre-trained models instead of custom CNN model. Trying 3 variants:

1. Pre-trained base + Train only fc - train new classification head for CIFAR-10 dataset
2. Fine-tune layer 4 + fc - train last layer & new classification head
3. Fine-tune full model - train entire model

```python
# Setup

!pip -q install torch torchvision --upgrade

import torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision, torchvision.transforms as T
from torchvision.models import resnet18, ResNet18_Weights
from torch.cuda.amp import autocast, GradScaler
import time, numpy as np

device  = torch.device("cuda" if torch.cuda.is_available() else "cpu")
USE_AMP = torch.cuda.is_available()
print("Device:", device)

# Data (use ImageNet-style transforms that match the pre-trained
weights)

weights = ResNet18_Weights.IMAGENET1K_V1
tfms = weights.transforms()

train_ds = torchvision.datasets.CIFAR10(root="./data2", train=True,
download=True, transform=tfms)
test_ds  = torchvision.datasets.CIFAR10(root="./data2", train=False,
download=True, transform=tfms)

BATCH_SIZE = 256
train_ld = DataLoader(train_ds,
                      batch_size=BATCH_SIZE,
                      shuffle=True,
                      num_workers=4,
                      persistent_workers=True,
                      prefetch_factor=2,
                      pin_memory=True
```

```
                     )
test_ld  = DataLoader(test_ds,
                      batch_size=BATCH_SIZE * 2,
                      shuffle=False,
                      num_workers=4,
                      persistent_workers=True,
                      prefetch_factor=2,
                      pin_memory=True
                     )
classes = train_ds.classes

# Model Evaluation

loss_fn = nn.CrossEntropyLoss()

@torch.no_grad()
def evaluate(model, loader):
  model.eval()
  loss_sum, correct, seen = 0.0, 0, 0

  for xb, yb in loader:
    xb = xb.to(device,
non_blocking=True).to(memory_format=torch.channels_last)
    yb = yb.to(device, non_blocking=True)

    with autocast(enabled=USE_AMP):
      logits = model(xb)
      loss = loss_fn(logits, yb)

    loss_sum = loss_sum + loss.item() * xb.size(0)
    correct = correct + (logits.argmax(1)==yb).sum().item()
    seen = seen + xb.size(0)

  return loss_sum / seen, correct / seen

def train_one_model(model, lr_optimizer, scheduler=None, epochs=10,
tag="exp"):
  USE_AMP = torch.cuda.is_available()
  scaler = GradScaler(enabled=USE_AMP)

  torch.backends.cudnn.benchmark = True
  model = model.to(device, memory_format=torch.channels_last)

  t0 = time.time()

  for ep in range(1, epochs+1):
    model.train()
    ls, corr, seen = 0.0, 0, 0

    for xb, yb in train_ld:
```

```
   xb = xb.to(device,
non_blocking=True).to(memory_format=torch.channels_last)
   yb = yb.to(device, non_blocking=True)

   lr_optimizer.zero_grad(set_to_none=True)

   with autocast(enabled=USE_AMP):
    logits = model(xb)
    loss = loss_fn(logits, yb)

   if USE_AMP:
    scaler.scale(loss).backward()
    scaler.step(lr_optimizer)
    scaler.update()
   else:
    loss.backward()
    lr_optimizer.step()

   ls = ls + loss.item() * xb.size(0)
   corr = corr + (logits.argmax(1)==yb).sum().item()
   seen = seen + xb.size(0)

  if scheduler: scheduler.step()

  train_loss, train_acc = ls / seen, corr / seen
  test_loss, test_acc = evaluate(model, test_ld)

  print(f"[{tag}] Epoch {ep:02d} | train {train_acc*100:5.2f}%
{train_loss:.4f} | test {test_acc*100:5.2f}% {test_loss:.4f}")
  print(f"[{tag}] total time: {time.time()-t0:.1f}s")
```

```
Device: cuda
```

```
100%|██████████| 170M/170M [00:06<00:00, 26.4MB/s]
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py
:627: UserWarning: This DataLoader will create 4 worker processes in
total. Our suggested max number of worker in current system is 2,
which is smaller than what this DataLoader is going to create. Please
be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential
slowness/freeze if necessary.
 warnings.warn(
```

```
# Experiment A: Train only fc

model = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
#print(model)

# replace head
model.fc = nn.Linear(model.fc.in_features, 10)
```

```python
# freeze everything except fc for training
for n, p in model.named_parameters():
  p.requires_grad = n.startswith("fc")

model = model.to(device)

lr_optimizer = torch.optim.AdamW(model.fc.parameters(), lr=1e-3,
weight_decay=1e-4)
lr_scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(lr_optimizer, T_max=10,
eta_min=1e-6)

train_one_model(model, lr_optimizer, scheduler=lr_scheduler,
epochs=10, tag="A_head_only")
```

```
Downloading: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth

100%|██████████| 44.7M/44.7M [00:00<00:00, 75.0MB/s]
/tmp/ipython-input-262670160.py:68: FutureWarning:
`torch.cuda.amp.GradScaler(args...)` is deprecated. Please use
`torch.amp.GradScaler('cuda', args...)` instead.
  scaler = GradScaler(enabled=USE_AMP)
/tmp/ipython-input-262670160.py:85: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):
/tmp/ipython-input-262670160.py:56: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):

[A_head_only] Epoch 01 | train 65.80% 1.0946 | test 74.45% 0.7873
[A_head_only] total time: 133.5s
[A_head_only] Epoch 02 | train 75.47% 0.7368 | test 76.33% 0.7057
[A_head_only] total time: 263.9s
[A_head_only] Epoch 03 | train 76.94% 0.6808 | test 77.24% 0.6770
[A_head_only] total time: 387.8s
[A_head_only] Epoch 04 | train 77.68% 0.6541 | test 77.43% 0.6602
[A_head_only] total time: 509.4s
[A_head_only] Epoch 05 | train 78.03% 0.6391 | test 77.91% 0.6522
[A_head_only] total time: 630.4s
[A_head_only] Epoch 06 | train 78.46% 0.6285 | test 78.07% 0.6473
[A_head_only] total time: 752.1s
[A_head_only] Epoch 07 | train 78.66% 0.6185 | test 77.98% 0.6428
[A_head_only] total time: 874.2s
[A_head_only] Epoch 08 | train 78.72% 0.6150 | test 78.20% 0.6405
[A_head_only] total time: 995.1s
[A_head_only] Epoch 09 | train 79.02% 0.6100 | test 78.46% 0.6372
```

```
[A_head_only] total time: 1116.7s
[A_head_only] Epoch 10 | train 79.13% 0.6093 | test 78.39% 0.6365
[A_head_only] total time: 1237.5s

# Experiment B: Train last layer + fc

model = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)

# Unfreeze layer4 + head
for p in model.parameters(): p.requires_grad = False
for p in model.layer4.parameters(): p.requires_grad = True
for p in model.fc.parameters():     p.requires_grad = True

# Lower LR for backbone, higher for head
param_groups = [
    {"params": model.layer4.parameters(), "lr": 5e-4, "weight_decay":
5e-5},
    {"params": model.fc.parameters(),     "lr": 1e-3, "weight_decay":
1e-4},
]
lr_optimizer = torch.optim.AdamW(param_groups)
lr_scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(lr_optimizer, T_max=20,
eta_min=1e-6)

model = model.to(device)
train_one_model(model, lr_optimizer, scheduler=lr_scheduler,
epochs=10, tag="B_last_layer")

Downloading: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth

100%|██████████| 44.7M/44.7M [00:00<00:00, 49.2MB/s]
/tmp/ipython-input-262670160.py:68: FutureWarning:
`torch.cuda.amp.GradScaler(args...)` is deprecated. Please use
`torch.amp.GradScaler('cuda', args...)` instead.
  scaler = GradScaler(enabled=USE_AMP)
/tmp/ipython-input-262670160.py:85: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):
/tmp/ipython-input-262670160.py:56: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):

[B_last_layer] Epoch 01 | train 83.32% 0.5763 | test 88.00% 0.3586
[B_last_layer] total time: 136.3s
[B_last_layer] Epoch 02 | train 94.57% 0.1588 | test 88.49% 0.3631
[B_last_layer] total time: 257.8s
```

```
[B_last_layer] Epoch 03 | train 98.18% 0.0525 | test 89.12% 0.4075
[B_last_layer] total time: 379.5s
[B_last_layer] Epoch 04 | train 98.77% 0.0360 | test 88.30% 0.4990
[B_last_layer] total time: 500.8s
[B_last_layer] Epoch 05 | train 98.98% 0.0287 | test 88.90% 0.4724
[B_last_layer] total time: 625.6s
[B_last_layer] Epoch 06 | train 99.47% 0.0168 | test 89.69% 0.4594
[B_last_layer] total time: 747.3s
[B_last_layer] Epoch 07 | train 99.67% 0.0109 | test 89.84% 0.4460
[B_last_layer] total time: 871.6s
[B_last_layer] Epoch 08 | train 99.86% 0.0052 | test 91.04% 0.4267
[B_last_layer] total time: 993.0s
[B_last_layer] Epoch 09 | train 99.97% 0.0019 | test 91.08% 0.4257
[B_last_layer] total time: 1117.3s
[B_last_layer] Epoch 10 | train 99.99% 0.0006 | test 91.38% 0.4151
[B_last_layer] total time: 1240.4s

# Experiment C: Train all layers (incl fc)

# ==== 0) Setup
================================================================
import time, os, torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision.transforms as T
from torchvision.datasets import CIFAR10
from torchvision.models import resnet18, ResNet18_Weights
from torch.cuda.amp import autocast, GradScaler

SEED = 42
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)
device  = torch.device("cuda" if torch.cuda.is_available() else "cpu")
USE_AMP = torch.cuda.is_available()
print("Device:", device)

# ==== 1) Data: CIFAR-10 with ImageNet-style preprocessing
====================
weights = ResNet18_Weights.IMAGENET1K_V1

# Use the weights' recommended normalization; add light augmentation
for train
_mean, _std = weights.transforms().mean, weights.transforms().std

train_tfms = T.Compose([
    T.Resize(256),
    T.RandomResizedCrop(224, scale=(0.7, 1.0)),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    T.Normalize(_mean, _std),
    T.RandomErasing(p=0.25),
])
test_tfms = T.Compose([
    T.Resize(224), T.CenterCrop(224),
    T.ToTensor(), T.Normalize(_mean, _std),
])

root = "./data3"
train_ds = CIFAR10(root, train=True,  download=True,
transform=train_tfms)
test_ds  = CIFAR10(root, train=False, download=True,
transform=test_tfms)
classes  = train_ds.classes

BATCH_SIZE = 128  # fits on Colab T4 with AMP; drop to 128 if OOM
n_workers = 2
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE,
shuffle=True,
                          num_workers=n_workers, pin_memory=True,
                          persistent_workers=False, prefetch_factor=2)
test_loader  = DataLoader(test_ds,  batch_size=BATCH_SIZE*2,
shuffle=False,
                          num_workers=n_workers, pin_memory=True,
                          persistent_workers=False, prefetch_factor=2)

# ==== 2) Model: ResNet-18 pretrained, replace head, unfreeze all
=============
model = resnet18(weights=weights)
model.fc = nn.Linear(model.fc.in_features, 10)  # CIFAR-10 classes

for p in model.parameters():
    p.requires_grad = True

# speed knobs
torch.backends.cudnn.benchmark = True
model = model.to(device, memory_format=torch.channels_last)

"""
try:
    model = torch.compile(model)  # PyTorch 2.x; if it errors, comment
it out
except Exception as e:
    print("torch.compile skipped:", e)
"""

# ==== 3) Optimizer: discriminative LRs + warmup→cosine
========================
# Smaller LR for early layers, larger for head. AdamW is a solid
default.
param_groups = [
    {"params": model.conv1.parameters(),  "lr": 1e-4, "weight_decay":
```

```
5e-5},
    {"params": model.bn1.parameters(),    "lr": 1e-4, "weight_decay":
0.0},
    {"params": model.layer1.parameters(), "lr": 1e-4, "weight_decay":
5e-5},
    {"params": model.layer2.parameters(), "lr": 2e-4, "weight_decay":
5e-5},
    {"params": model.layer3.parameters(), "lr": 3e-4, "weight_decay":
5e-5},
    {"params": model.layer4.parameters(), "lr": 3e-4, "weight_decay":
5e-5},
    {"params": model.fc.parameters(),     "lr": 1e-3, "weight_decay":
1e-4},  # head highest
]
optimizer = torch.optim.AdamW(param_groups)

EPOCHS = 10
WARMUP_E = 3
scheduler = torch.optim.lr_scheduler.SequentialLR(
    optimizer,
    schedulers=[
        torch.optim.lr_scheduler.LinearLR(optimizer, start_factor=0.1,
total_iters=WARMUP_E),
        torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=EPOCHS - WARMUP_E, eta_min=1e-6),
    ],
    milestones=[WARMUP_E],
)

loss_fn = nn.CrossEntropyLoss(label_smoothing=0.05)
scaler  = GradScaler(enabled=USE_AMP)

# ==== 4) Eval helper
==========================================================
@torch.no_grad()
def evaluate(model, loader):
    model.eval()
    loss_sum, correct, seen = 0.0, 0, 0
    for xb, yb in loader:
        xb = xb.to(device,
non_blocking=True).to(memory_format=torch.channels_last)
        yb = yb.to(device, non_blocking=True)

        with autocast(enabled=USE_AMP):
            logits = model(xb)
            loss   = loss_fn(logits, yb)

        bs = xb.size(0)
        loss_sum += loss.item() * bs
        correct  += (logits.argmax(1) == yb).sum().item()
        seen     += bs
    return loss_sum/seen, correct/seen

# ==== 5) Train loop
===========================================================
def my_train(model, train_loader, test_loader):
    best_acc, best_state = 0.0, None
    t0_all = time.time()

    for epoch in range(1, EPOCHS+1):
        model.train()
        loss_sum, correct, seen = 0.0, 0, 0
        t0 = time.time()

        for xb, yb in train_loader:
            xb = xb.to(device,
non_blocking=True).to(memory_format=torch.channels_last)
            yb = yb.to(device, non_blocking=True)

            optimizer.zero_grad(set_to_none=True)
            with autocast(enabled=USE_AMP):
                logits = model(xb)
                loss   = loss_fn(logits, yb)

            if USE_AMP:
                scaler.scale(loss).backward()
                # optional: clip for stability on full FT
                scaler.unscale_(optimizer)
                torch.nn.utils.clip_grad_norm_(model.parameters(),
1.0)
                scaler.step(optimizer); scaler.update()
            else:
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(),
1.0)
                optimizer.step()

            bs = xb.size(0)
            loss_sum += loss.item() * bs
            correct  += (logits.argmax(1) == yb).sum().item()
            seen     += bs

        scheduler.step()
        train_loss, train_acc = loss_sum/seen, correct/seen
        val_loss, val_acc = evaluate(model, test_loader)

        # track best
        if val_acc > best_acc:
            best_acc  = val_acc
            best_state = {k: v.detach().cpu() for k, v in
```

```
model.state_dict().items()}

        # log
        cur_lr = optimizer.param_groups[-1]["lr"]  # head LR just to
print
        print(f"Epoch {epoch:02d}/{EPOCHS} | "
              f"train {train_acc*100:5.2f}% loss {train_loss:.4f} | "
              f"val {val_acc*100:5.2f}% loss {val_loss:.4f} | "
              f"lr {cur_lr:.6f} | {time.time()-t0:.1f}s")

    # restore best and save
    if best_state is not None:
        model.load_state_dict(best_state)

    os.makedirs("checkpoints", exist_ok=True)
    torch.save(model.state_dict(),
"checkpoints/resnet18_cifar10_fullfinetune_best.pth")
    print(f"Best val acc: {best_acc*100:.2f}% | total time:
{time.time()-t0_all:.1f}s")
    return model

model = my_train(model, train_loader, test_loader)

# ==== 6) Tiny inference helper
===============================================
@torch.no_grad()
def predict_one(img_tensor):  # img_tensor should be already
transformed
    model.eval()
    x =
img_tensor.unsqueeze(0).to(device).to(memory_format=torch.channels_las
t)

    with autocast(enabled=USE_AMP):
        logits = model(x)
        probs  = F.softmax(logits, dim=1).squeeze(0).cpu()

    idx = int(torch.argmax(probs))
    return classes[idx], float(probs[idx])

# Example:
img, label = test_ds[0]
pred, p = predict_one(img)
print("Actual:", classes[label], "| Pred:", pred, f"({p:.2%})")
```

```
Device: cuda
```

```
/tmp/ipython-input-3889634809.py:94: FutureWarning:
`torch.cuda.amp.GradScaler(args...)` is deprecated. Please use
`torch.amp.GradScaler('cuda', args...)` instead.
```

```
  scaler  = GradScaler(enabled=USE_AMP)
/tmp/ipython-input-3889634809.py:130: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):
/tmp/ipython-input-3889634809.py:105: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast(enabled=USE_AMP):

Epoch 01/10 | train 77.88% loss 0.8863 | val 89.99% loss 0.5822 | lr
0.000400 | 177.5s
Epoch 02/10 | train 89.70% loss 0.5777 | val 93.21% loss 0.4924 | lr
0.000700 | 168.7s

/usr/local/lib/python3.12/dist-packages/torch/optim/
lr_scheduler.py:209: UserWarning: The epoch parameter in
`scheduler.step()` was not necessary and is being deprecated where
possible. Please use `scheduler.step()` to step the scheduler. During
the deprecation, if epoch is different from None, the closed form is
used instead of the new chainable form, where available. Please open
an issue if you are unable to replicate your use case:
https://github.com/pytorch/pytorch/issues/new/choose.
  warnings.warn(EPOCH_DEPRECATION_WARNING, UserWarning)

Epoch 03/10 | train 91.61% loss 0.5256 | val 92.54% loss 0.5087 | lr
0.001000 | 171.6s
Epoch 04/10 | train 92.13% loss 0.5100 | val 92.81% loss 0.4894 | lr
0.000951 | 168.2s
Epoch 05/10 | train 93.98% loss 0.4645 | val 92.91% loss 0.4858 | lr
0.000812 | 168.7s
```