This can be run run on Google Colab using this link

CIFAR-10 Classification (Fully-Connected vs. Convolutional)

In this notebook, we will:

1. Download **CIFAR-10** (a dataset of 32×32 color images in 10 classes).
2. Demonstrate a working classifier using **fully-connected (FC) layers** (a simple MLP).
3. **Exercise**: Students will create a **convolutional** version for better efficiency.
4. Compare **parameter counts** and performance.

This exercise is just an opportunity to understand the power of weight-sharing and play with a standard classification setting that for decades was a focus of machine learning researchers.

Try to improve the test performance of the network without making it more expensive to train. You will just be graded in your experiment findings at the end.

Key Points:

- CIFAR-10 has 60,000 images (50k train, 10k test).
- Each image is 3×32×32 (3 color channels).
- We'll flatten those 3×32×32 = 3072 pixels as input to a fully-connected MLP.
- Then we'll invite you to use convolutional layers, which drastically reduce parameters by sharing weights.

---

1. Setup

We'll import **PyTorch**, **torchvision**, then load CIFAR-10. We'll make small transformations (convert to tensors, normalize if desired).

In [7]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as T
import numpy as np

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

# Basic transforms: ToTensor (range [0,1]), optional normalization.
transform = T.Compose(
    [
        T.ToTensor(),
```

```
        # Optionally normalize: T.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))
    ]
)

# Download and create datasets
train_dataset = torchvision.datasets.CIFAR10(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.CIFAR10(
    root="./data", train=False, download=True, transform=transform
)

# Dataloaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True, num_workers=2
)
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=batch_size, shuffle=False, num_workers=2
)
```

```
Using device: cuda
Files already downloaded and verified
Files already downloaded and verified
```

2. A Simple Fully-Connected (MLP) Classifier

We'll define a basic MLP:

1. Flatten the 3×32×32 image (3072 dims).
2. Several **fully connected layers**, then 10 outputs (one per CIFAR-10 class).

We can train it for a few epochs—**this won't achieve high accuracy** (CNNs do much better), but it demonstrates the approach.

In [8]:
```python
class SimpleMLP(nn.Module):
    def __init__(self, input_dim=3 * 32 * 32, hidden_dim=100, num_classes=10):
        super().__init__()
        # A small 2-layer MLP:
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        # x: shape (batch, 3, 32, 32)
        batch_size = x.size(0)
        x = x.view(batch_size, -1)  # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


mlp = SimpleMLP().to(device)
print(
    "MLP parameter count:", sum(p.numel() for p in mlp.parameters() if p.requires_grad)
)
```

```
MLP parameter count: 308310
```

2.1 Training Loop

We define a simple function `train_epoch` and `test_accuracy` to measure performance.

In [9]:
```python
import torch.optim as optim


def train_epoch(model, loader, optimizer, loss_fn=nn.CrossEntropyLoss()):
    model.train()
    total_loss = 0.0
    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        preds = model(images)
        loss = loss_fn(preds, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(loader)


def test_accuracy(model, loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            preds = model(images)
            predicted = preds.argmax(dim=1)
            correct += (predicted == labels).sum().item()
```

```
            total += labels.size(0)
    return 100.0 * correct / total
```

Now let's do a short training run on the MLP—**note** that this won't get anywhere close to SOTA accuracy on CIFAR-10, but it demonstrates the pipeline. We'll do maybe **2** or **3** epochs just to see it learns something.

In [23]:
```
mlp = SimpleMLP().to(device)
optimizer = optim.Adam(mlp.parameters(), lr=1e-3)

epochs = 3  # can increase if you want
for epoch in range(1, epochs + 1):
    train_loss = train_epoch(mlp, train_loader, optimizer)
    test_acc = test_accuracy(mlp, test_loader)
    print(
        f"Epoch {epoch}/{epochs}, train loss={train_loss:.4f}, test acc={test_acc:.2f}%"
    )
```

```
Epoch 1/3, train loss=1.8783, test acc=36.24%
Epoch 2/3, train loss=1.7282, test acc=41.13%
Epoch 3/3, train loss=1.6665, test acc=42.35%
```

3. Exercise: Use a Stack of Convolutions

CIFAR-10 was **designed** with 2D images in mind, so we can do **far better** with **convolutional** layers that share weights locally.

Your Tasks

1. **Construct** a new network (say `ConvNet`) with multiple convolutional layers, optional pooling, etc.
2. **Count** the number of parameters. *(Hint:* `sum(p.numel() for p in model.parameters() if p.requires_grad)` *.)*
3. **Train** this model on CIFAR-10. Try to achieve comparable or better accuracy than the MLP **with fewer parameters.**

Suggested Skeleton Code

Below is a minimal skeleton. Feel free to modify layer dimensions, add pooling, or add more conv layers. We provide the class structure for you to fill in.

In [49]:
```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # define your convolutional layers here.
        # e.g.
        # self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1)
        # self.pool = nn.MaxPool2d(2,2)
        # etc.
        # Then define a final linear layer.
        # You have to figure out the shape after the conv layers.
```

```
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        # after 2 conv+pool steps, etc...
        # But let's suppose we do only 1 pool, etc.

        self.fc = nn.Linear(16 * 16 * 16, num_classes)  # Just a guess of dimensions.

    def forward(self, x):
        # x: (batch, 3, 32, 32)
        x = F.relu(self.conv1(x))  # (batch,8,32,32)
        x = self.pool(F.relu(self.conv2(x)))  # (batch,16,16,16)
        # flatten
        batch_size = x.size(0)
        x = x.view(batch_size, -1)
        x = self.fc(x)
        return x
```

In [50]:
```
# STUDENT EXERCISE:
convnet = ConvNet().to(device)
print(
    "ConvNet param count:",
    sum(p.numel() for p in convnet.parameters() if p.requires_grad),
)

optimizer_conv = optim.Adam(convnet.parameters(), lr=1e-3)
epochs_conv = 3
for epoch in range(1, epochs_conv + 1):
    train_loss = train_epoch(convnet, train_loader, optimizer_conv)
    test_acc = test_accuracy(convnet, test_loader)
    print(
        f"[ConvNet] Epoch {epoch}/{epochs_conv}, train loss={train_loss:.4f}, test acc=
{test_acc:.2f}%"
    )

# print(
#     "\nNow consider adjusting your ConvNet architecture, parameter count, etc. for better
results."
# )
```

```
ConvNet param count: 42362
[ConvNet] Epoch 1/3, train loss=1.5448, test acc=53.37%
[ConvNet] Epoch 2/3, train loss=1.2001, test acc=60.35%
[ConvNet] Epoch 3/3, train loss=1.0696, test acc=61.75%
```

3.1 Code: Train Your ConvNet

Exercise: Implement the training loop (similar to the MLP), measure test accuracy, and see how you can reduce or increase parameters to trade off accuracy vs. model size.

Examples:

- Add more conv layers or channels.
- Add more or fewer pooling layers.

- Print out the param count.
- Play with other architectural tricks such as residual connections.
- Tweak the learning rate or optimizer.

Try to see how low you can go in param count while maintaining a decent accuracy!

Plain ConvNet

In [46]:
```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # define your convolutional layers here.
        # e.g.
        # self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1)
        # self.pool = nn.MaxPool2d(2,2)
        # etc.
        # Then define a final linear layer.
        # You have to figure out the shape after the conv layers.
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.avgpool = nn.AvgPool2d(2, 2)

        self.fc = nn.Linear(64 * 8 * 8, num_classes)

        # after 2 conv+pool steps, etc...
        # But let's suppose we do only 1 pool, etc.

    def forward(self, x):
        # x: (batch, 3, 32, 32)
        x = F.relu(self.conv1(x))  # (batch,8,32,32)
        x = F.relu(self.conv2(x))  # (batch,8,32,32)
        x = self.pool1(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.relu(self.conv5(x))
        x = self.avgpool(x)

        # flatten
        batch_size = x.size(0)
        x = x.view(batch_size, -1)
        x = self.fc(x)
        return x
```

In [47]:
```
# STUDENT EXERCISE:
convnet = ConvNet().to(device)
print(
    "ConvNet param count:",
    sum(p.numel() for p in convnet.parameters() if p.requires_grad),
)

optimizer_conv = optim.Adam(convnet.parameters(), lr=1e-3)
```

```
epochs_conv = 3
for epoch in range(1, epochs_conv + 1):
    train_loss = train_epoch(convnet, train_loader, optimizer_conv)
    test_acc = test_accuracy(convnet, test_loader)
    print(
        f"[ConvNet] Epoch {epoch}/{epochs_conv}, train loss={train_loss:.4f}, test acc=
{test_acc:.2f}%"
    )

# print(
#     "\nNow consider adjusting your ConvNet architecture, parameter count, etc. for better
results."
# )
```

```
ConvNet param count: 143466
[ConvNet] Epoch 1/3, train loss=1.5962, test acc=52.24%
[ConvNet] Epoch 2/3, train loss=1.1431, test acc=63.08%
[ConvNet] Epoch 3/3, train loss=0.9626, test acc=66.47%
```

Minimize # of param while keeping similar acc with bottleneck resblocks

In [34]:

```
class BottleneckResBlock(nn.Module):
    def __init__(self, in_channels, bottleneck_channels, out_channels):
        """
        Args:
            in_channels (int): Number of input channels.
            bottleneck_channels (int): Number of channels in the middle convs.
            out_channels (int): Number of output channels.
        """
        super(BottleneckResBlock, self).__init__()

        # 1x1 convolution
        self.conv1 = nn.Conv2d(
            in_channels, bottleneck_channels, kernel_size=1, bias=False
        )
        self.bn1 = nn.BatchNorm2d(bottleneck_channels)

        # 3x3 convolution
        self.conv2 = nn.Conv2d(
            bottleneck_channels,
            bottleneck_channels,
            kernel_size=3,
            padding=1,
            bias=False,
        )
        self.bn2 = nn.BatchNorm2d(bottleneck_channels)

        # 1x1 convolution
        self.conv3 = nn.Conv2d(
            bottleneck_channels, out_channels, kernel_size=1, bias=False
        )
        self.bn3 = nn.BatchNorm2d(out_channels)

        # ReLU
        self.relu = nn.ReLU(inplace=True)

        # If in_channels != out_channels, we need a projection (1x1) for the skip
```

```
        if in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.shortcut = nn.Identity()  # Use `Identity` if dimensions match

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        # Apply shortcut if needed
        identity = self.shortcut(identity)

        # Residual addition
        out += identity

        # Final ReLU
        out = self.relu(out)

        return out
```

In [51]:

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # define your convolutional layers here.
        # e.g.
        # self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1)
        # self.pool = nn.MaxPool2d(2,2)
        # etc.
        # Then define a final linear layer.
        # You have to figure out the shape after the conv layers.

        self.block1 = BottleneckResBlock(3, 8, 16)
        self.block2 = BottleneckResBlock(16, 8, 16)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.block3 = BottleneckResBlock(16, 16, 32)
        self.block4 = BottleneckResBlock(32, 16, 32)
        self.block5 = BottleneckResBlock(32, 16, 32)
        self.avgpool = nn.AdaptiveAvgPool2d((8, 8))

        # after 2 conv+pool steps, etc...
        # But let's suppose we do only 1 pool, etc.

        self.fc = nn.Linear(32 * 8 * 8, num_classes)
```

```
    def forward(self, x):
        # x: (batch, 3, 32, 32)
        x = self.block1(x)  # (batch, 16, 32, 32)
        x = self.block2(x)  # (batch, 16, 32, 32)
        x = self.pool1(x)  # (batch, 16, 16, 16)
        x = self.block3(x)  # (batch, 32, 16, 16)
        x = self.block4(x)  # (batch, 32, 16, 16)
        x = self.block5(x)  # (batch, 32, 16, 16)
        x = self.avgpool(x)  # (batch, 32, 8, 8)

        # flatten
        batch_size = x.size(0)
        x = x.view(batch_size, -1)
        x = self.fc(x)
        return x
```

In [52]:
```
# STUDENT EXERCISE:
convnet = ConvNet().to(device)
print(
    "ConvNet param count:",
    sum(p.numel() for p in convnet.parameters() if p.requires_grad),
)

optimizer_conv = optim.Adam(convnet.parameters(), lr=1e-3)
epochs_conv = 3
for epoch in range(1, epochs_conv + 1):
    train_loss = train_epoch(convnet, train_loader, optimizer_conv)
    test_acc = test_accuracy(convnet, test_loader)
    print(
        f"[ConvNet] Epoch {epoch}/{epochs_conv}, train loss={train_loss:.4f}, test acc=
{test_acc:.2f}%"
    )

# print(
#     "\nNow consider adjusting your ConvNet architecture, parameter count, etc. for better
results."
# )
```

```
ConvNet param count: 32946
[ConvNet] Epoch 1/3, train loss=1.4450, test acc=56.19%
[ConvNet] Epoch 2/3, train loss=1.0936, test acc=62.19%
[ConvNet] Epoch 3/3, train loss=0.9506, test acc=66.62%
```

Maximizing Acc while keeping # of param around the same

In [59]:
```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # define your convolutional layers here.
        # e.g.
        # self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1)
        # self.pool = nn.MaxPool2d(2,2)
        # etc.
        # Then define a final linear layer.
        # You have to figure out the shape after the conv layers.
```

```
        self.block1 = BottleneckResBlock(3, 16, 32)
        self.block2 = BottleneckResBlock(32, 16, 32)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.block3 = BottleneckResBlock(32, 32, 64)
        self.block4 = BottleneckResBlock(64, 32, 64)
        self.block5 = BottleneckResBlock(64, 64, 64)
        self.avgpool = nn.AdaptiveAvgPool2d((8, 8))

        # after 2 conv+pool steps, etc...
        # But let's suppose we do only 1 pool, etc.

        self.fc = nn.Linear(64 * 8 * 8, num_classes)

    def forward(self, x):
        # x: (batch, 3, 32, 32)
        x = self.block1(x)  # (batch, 32, 32, 32)
        x = self.block2(x)  # (batch, 32, 32, 32)
        x = self.pool1(x)  # (batch, 32, 16, 16)
        x = self.block3(x)  # (batch, 64, 16, 16)
        x = self.block4(x)  # (batch, 64, 16, 16)
        x = self.block5(x)  # (batch, 64, 16, 16)
        x = self.avgpool(x)  # (batch, 64, 8, 8)

        # flatten
        batch_size = x.size(0)
        x = x.view(batch_size, -1)
        x = self.fc(x)
        return x
```

In [60]:

```
# STUDENT EXERCISE:
convnet = ConvNet().to(device)
print(
    "ConvNet param count:",
    sum(p.numel() for p in convnet.parameters() if p.requires_grad),
)

optimizer_conv = optim.Adam(convnet.parameters(), lr=1e-3)
epochs_conv = 3
for epoch in range(1, epochs_conv + 1):
    train_loss = train_epoch(convnet, train_loader, optimizer_conv)
    test_acc = test_accuracy(convnet, test_loader)
    print(
        f"[ConvNet] Epoch {epoch}/{epochs_conv}, train loss={train_loss:.4f}, test acc=
{test_acc:.2f}%"
    )

# print(
#     "\nNow consider adjusting your ConvNet architecture, parameter count, etc. for better
results."
# )
```

```
ConvNet param count: 121306
[ConvNet] Epoch 1/3, train loss=1.3278, test acc=65.04%
[ConvNet] Epoch 2/3, train loss=0.8844, test acc=69.51%
[ConvNet] Epoch 3/3, train loss=0.7386, test acc=73.01%
```

4. Report Your Findings

Points to understand:

1. A **fully-connected** approach to image classification (such as CIFAR-10) can work but tends to have **many** parameters (e.g., 3,072×100 just in one layer on tiny images) and typically yields lower accuracy compared to modern **Convolutional** architectures.
2. **Convolution** drastically reduces parameter counts via **weight sharing**, can often achieve much higher accuracy on image tasks, and is typically *translation-equivariant*.
3. Your goal is to **experiment** with different conv net designs to minimize param count while maximizing accuracy.

Report here at least two iterations of your architectural experiments:

1. Using an architecture consisting of $\boxed{\text{convolution, max pool, average pool}}$, I was able to reduce the parameterization to $\boxed{143466}$ parameters and achieve test accuracy of $\boxed{66.47\%}$ after three epochs of training.

2. In a second test, I tried an architecture consisting of $\boxed{\text{all above} + \text{residual bottleneck blocks} + \text{batch norm}}$. That used an even smaller parameterization, with only $\boxed{32946}$ parameters, and it achieved test accuracy of $\boxed{66.62\%}$ after three epochs of training.

3. In the third test, I tried to increase the dimension of residual blocks, using $\boxed{121306}$ parameters, and it achieved test accuracy of $\boxed{73.01\%}$ after three epochs of training.

Good luck!