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.

---

0. DataLoader Fundamentals: Handling Large-Scale Data Efficiently

So far, we've worked with small datasets that can be loaded directly into a single tensor (like in our optimization exercises). However, most real-world datasets, especially in image processing, are far too large to fit into your computer's RAM all at once. Imagine trying to load all 14 million images of the ImageNet dataset—it's impossible for most machines!

The Problem:

- CIFAR-10 has 60,000 images (50k train + 10k test)
- Each image is 32×32×3 = 3,072 values
- Loading all at once: 50,000 × 3,072 × 4 bytes ≈ 600MB (manageable)
- But ImageNet? 14M × 224×224×3 × 4 bytes ≈ 8.4TB! ▯

The Solution: PyTorch's **DataLoader** - a powerful abstraction that:

1. Loads data in small **batches** (e.g., 64 images at a time)
2. **Shuffles** data each epoch for better training
3. Performs **on-the-fly transformations** (augmentation, normalization)
4. Loads data in **parallel** using multiple CPU workers
5. Automatically handles the **iteration** over your dataset

Let's build intuition step by step:

0.1 The Dataset Class - Your Data Container

```python
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np

# Let's create a toy dataset to understand the mechanics
class ToyDataset(Dataset):
    """A simple dataset of random 'images' and labels"""
    def __init__(self, num_samples=1000, image_size=(3, 32, 32)):
        # In reality, you'd load file paths or data here
        self.num_samples = num_samples
        self.image_size = image_size

    def __len__(self):
        """Tell PyTorch how many samples we have"""
        return self.num_samples

    def __getitem__(self, idx):
        """Return one sample at index idx
        This is called by the DataLoader to fetch data"""
        # Simulate loading an image (in practice, you'd read from 
disk)
        fake_image = torch.randn(self.image_size)  # Random "image"
        fake_label = torch.randint(0, 10, (1,)).item()  # Random label 
(0-9)
        return fake_image, fake_label

# Create our toy dataset
toy_dataset = ToyDataset(num_samples=1000)
print(f"Dataset has {len(toy_dataset)} samples")
print(f"Sample 0 shape: image={toy_dataset[0][0].shape}, 
label={toy_dataset[0][1]}")
```

```
Dataset has 1000 samples
Sample 0 shape: image=torch.Size([3, 32, 32]), label=7
```

0.2 The DataLoader - Your Batch Manager

Now that we have a Dataset, let's see how DataLoader transforms it into efficient batches:

```python
# Create a DataLoader with different settings
toy_loader = DataLoader(
    toy_dataset,
    batch_size=32,      # Process 32 samples at once
    shuffle=True,       # Randomize order each epoch
    num_workers=0       # Use main process (set >0 for parallel
loading)
)

# Let's examine what the DataLoader gives us
for batch_idx, (images, labels) in enumerate(toy_loader):
    print(f"Batch {batch_idx}: images shape={images.shape}, labels
shape={labels.shape}")
    if batch_idx == 2:  # Just show first 3 batches
        break

print(f"\nTotal batches: {len(toy_loader)} = {len(toy_dataset)}
samples ÷ {32} batch_size")
```

```
Batch 0: images shape=torch.Size([32, 3, 32, 32]), labels
shape=torch.Size([32])
Batch 1: images shape=torch.Size([32, 3, 32, 32]), labels
shape=torch.Size([32])
Batch 2: images shape=torch.Size([32, 3, 32, 32]), labels
shape=torch.Size([32])

Total batches: 32 = 1000 samples ÷ 32 batch_size
```

0.3 Key Takeaways

□ **Dataset** defines how to access your data (one sample at a time) □ **DataLoader** = Batches multiple samples together efficiently □ **Batch size** = How many samples to process together (larger = more stable gradients) □ **Shuffle** = Randomizes order each epoch; prevents the model from memorizing data order (crucial for good training!) □ **num_workers** = Parallel CPU threads for loading (*0* = main thread, *>0* = faster loading, but more RAM)

Memory Rule of Thumb:

- Batch size × Sample size × 4 bytes (float32) × ~3 (for gradients) = GPU memory needed
- Example: 64 images × 3×224×224 × 4 bytes × 3 ≈ 115MB per batch

Why this matters:

- **Memory efficient**: Only loads what fits in GPU memory
- **Better gradients**: Averaging over batches reduces noise
- **Faster training**: GPU parallelism works best with batches
- **Shuffling prevents overfitting**: Model can't memorize data order

In our main assignment code, `torchvision.datasets.CIFAR10(...)` is simply a more complex, pre-built `Dataset` provided by PyTorch that knows how to load CIFAR-10 images from disk one by one. The `DataLoader` then wraps it to give us the shuffled mini-batches we need for training.

Now that we understand how the data is loaded, let's set up the pipeline for our CIFAR-10 experiment.

---

1. Setup

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

```python
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
```

```
100%|██████████| 170M/170M [00:12<00:00, 13.2MB/s]
```

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.

```
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.

```
import torch.optim as optim

def train_epoch(model, loader, optimizer,
loss_fn=nn.CrossEntropyLoss()):
    model.train()
    total_loss = 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.

```
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=1.8844, test acc=37.83%
Epoch 2/3, train loss=1.7170, test acc=41.22%
Epoch 3/3, train loss=1.6539, test acc=42.17%
```

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

Task 3.1 (3 points): Construct a new network (say `ConvNet`) with multiple convolutional layers, optional pooling, etc.

Task 3.2 (1 point): Count the number of parameters. *(Hint: `sum(p.numel() for p in model.parameters() if p.requires_grad)`.)*

Task 3.3 (2 points): 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.

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # Conv layer 1: 3 input channels (RGB), 8 output feature maps,
3x3 kernel, padding=1 preserves spatial dims
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, padding=1)
        # Conv layer 2: 8 input channels, 16 output feature maps, 3x3
kernel, padding=1 preserves spatial dims
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        # Max pooling: 2x2 window, stride 2 — halves spatial
dimensions
        self.pool = nn.MaxPool2d(2, 2)
        # Fully connected output layer: flattened feature map -> 10
classes
        # After conv1 (32x32) -> conv2+pool (16x16): feature map is 16
x 16 x 16 = 4096
        self.fc = nn.Linear(16 * 16 * 16, num_classes)

    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)
        batch_size = x.size(0)
        x = x.view(batch_size, -1)           # -> (batch, 4096)
        x = self.fc(x)                       # -> (batch, 10)
        return x
```

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!

```
# 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=1.5613, test acc=52.73%
[ConvNet] Epoch 2/3, train loss=1.2365, test acc=57.39%
[ConvNet] Epoch 3/3, train loss=1.1339, test acc=59.92%

Now consider adjusting your ConvNet architecture, parameter count,
etc. for better results.
```

```
class ConvNetSmall(nn.Module):
    """A smaller ConvNet with reduced channels and two pooling layers.

    Architecture:
        Conv2d(3, 4, 3, padding=1)  -> ReLU -> MaxPool2d(2,2)  => (4,
16, 16)
        Conv2d(4, 8, 3, padding=1)  -> ReLU -> MaxPool2d(2,2)  => (8,
8, 8)
        Linear(8*8*8, 10)

    This aggressively reduces spatial dimensions via two pooling
layers,
    and uses half the channels of the original ConvNet, shrinking the
    dominant FC layer from 4096->10 down to 512->10.
    """
    def __init__(self, num_classes=10):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 4, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        # After conv1+pool: (4, 16, 16)
        # After conv2+pool: (8, 8, 8) -> flatten = 512
        self.fc = nn.Linear(8 * 8 * 8, num_classes)

    def forward(self, x):
        # x: (batch, 3, 32, 32)
        x = self.pool(F.relu(self.conv1(x)))  # -> (batch, 4, 16, 16)
        x = self.pool(F.relu(self.conv2(x)))  # -> (batch, 8, 8, 8)
        x = x.view(x.size(0), -1)             # -> (batch, 512)
        x = self.fc(x)                        # -> (batch, 10)
        return x
```

```
class ConvNetTiny(nn.Module):
    """An even tinier ConvNet with three conv layers and three pooling
stages.

    Architecture:
        Conv2d(3, 4, 3, padding=1)  -> ReLU -> MaxPool2d(2,2)  => (4,
16, 16)
        Conv2d(4, 8, 3, padding=1)  -> ReLU -> MaxPool2d(2,2)  => (8,
8, 8)
        Conv2d(8, 16, 3, padding=1) -> ReLU -> MaxPool2d(2,2)  => (16,
4, 4)
        Linear(16*4*4, 10)

    Three pooling layers reduce 32x32 -> 16x16 -> 8x8 -> 4x4,
    so the FC layer only maps 256 -> 10.
    """
    def __init__(self, num_classes=10):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 4, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        # After three pool stages: 32 -> 16 -> 8 -> 4, with 16
channels => 16*4*4 = 256
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # -> (batch, 4, 16, 16)
        x = self.pool(F.relu(self.conv2(x)))  # -> (batch, 8, 8, 8)
        x = self.pool(F.relu(self.conv3(x)))  # -> (batch, 16, 4, 4)
        x = x.view(x.size(0), -1)             # -> (batch, 256)
        x = self.fc(x)                        # -> (batch, 10)
        return x

# --- Experiment 2: Smaller ConvNet ---
convnet_small = ConvNetSmall().to(device)
print("Small ConvNet param count:", sum(p.numel() for p in
convnet_small.parameters() if p.requires_grad))

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

```
Small ConvNet param count: 5538
[SmallConvNet] Epoch 1/3, train loss=1.8019, test acc=44.85%
```

```
[SmallConvNet] Epoch 2/3, train loss=1.5290, test acc=46.85%
[SmallConvNet] Epoch 3/3, train loss=1.4569, test acc=49.64%
```

```python
# --- Experiment 3: Tiny ConvNet ---
convnet_tiny = ConvNetTiny().to(device)
print("Tiny ConvNet param count:", sum(p.numel() for p in
convnet_tiny.parameters() if p.requires_grad))

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

```
Tiny ConvNet param count: 4146
[TinyConvNet] Epoch 1/3, train loss=1.8401, test acc=42.49%
[TinyConvNet] Epoch 2/3, train loss=1.5407, test acc=46.70%
[TinyConvNet] Epoch 3/3, train loss=1.4530, test acc=50.26%
```

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.

Task 4.1 (2 points): Report here at least two iterations of your architectural experiments:

1. Using an architecture consisting of **two convolutional layers (Conv2d(3,8,3,padding=1) → ReLU → Conv2d(8,16,3,padding=1) → ReLU → MaxPool2d(2,2)) followed by a fully connected layer (Linear(4096, 10))**, I was able to reduce the parameterization to **42,362** parameters and achieve test accuracy of **59.92%** after three epochs of training.

2. In the second test, I tried an architecture consisting of **three convolutional layers with pooling after each (Conv2d(3,4,3,padding=1) → ReLU → Pool → Conv2d(4,8,3,padding=1) → ReLU → Pool → Conv2d(8,16,3,padding=1) → ReLU → Pool) followed by a fully connected layer (Linear(256, 10))**. That used an even smaller parameterization, with only **4,146** parameters, and it achieved test accuracy of **50.26%** after three epochs of training.

3. In the third test, I tried an architecture consisting of **two convolutional layers with reduced channels and pooling after each (Conv2d(3,4,3,padding=1) → ReLU → Pool → Conv2d(4,8,3,padding=1) → ReLU → Pool) followed by a fully connected layer (Linear(512, 10))**. That used a parameterization of **5,538** parameters and achieved test accuracy of **49.64%** after three epochs of training.