In [ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Verificar si CUDA está disponible y configurar el dispositivo (utilizando la GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

# Definir la arquitectura de AlexNet
class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            # Capa convolucional Conv1 con ReLU
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            # Capa de Max Pooling Pool1
            nn.MaxPool2d(kernel_size=3, stride=2),
            # Capa convolucional Conv2 con ReLU
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            # Capa de Max Pooling Pool2
            nn.MaxPool2d(kernel_size=3, stride=2),
            # Capas convolucionales Conv3, Conv4, Conv5 con ReLU
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            # Capa de Max Pooling Pool5
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.classifier = nn.Sequential(
            # Capa de Dropout
            nn.Dropout(),
            # Capa totalmente conectada (FC6) con ReLU
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            # Capa de Dropout
            nn.Dropout(),
            # Capa totalmente conectada (FC7) con ReLU
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            # Capa de salida con num_classes (en este caso, 10 para CIFAR-10)
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
```

```
        x = self.classifier(x)
        return x
```

```
cuda:0
```

In [ ]:
```
# Descargar y cargar el conjunto de datos CIFAR-10
transform = transforms.Compose([transforms.Resize((224, 224)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, t
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True, nu

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False, num
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [ ]:
```
# Inicializar la red y el optimizador en la GPU si está disponible
net = AlexNet().to(device)  # Mueve la red AlexNet a la GPU si está disponible
criterion = nn.CrossEntropyLoss()  # Función de pérdida de entropía cruzada
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)  # Optimizador SGD c
```

In [ ]:
```
# Entrenar la red
for epoch in range(10):  # Número de épocas de entrenamiento (puedes ajustarlo)
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)  # Mover datos a la G
        optimizer.zero_grad()  # Reiniciar los gradientes acumulados
        outputs = net(inputs)  # Propagar hacia adelante (forward pass)
        loss = criterion(outputs, labels)  # Calcular la pérdida
        loss.backward()  # Retropropagación (backpropagation)
        optimizer.step()  # Actualizar los pesos de la red
        running_loss += loss.item()
    print(f'Epoch {epoch + 1}, Loss: {running_loss / (i + 1)}')  # Imprimir la pérd

print('Entrenamiento finalizado')  # Mensaje de finalización del entrenamiento
```

```
Epoch 1, Loss: 1.7963611483192565
Epoch 2, Loss: 1.1888328436392663
Epoch 3, Loss: 0.926413688935001
Epoch 4, Loss: 0.7821292494514853
Epoch 5, Loss: 0.6686124411707723
Epoch 6, Loss: 0.5863529806879187
Epoch 7, Loss: 0.527399187281928
Epoch 8, Loss: 0.47528276032865313
Epoch 9, Loss: 0.4373498583928713
Epoch 10, Loss: 0.3973131573968165
Entrenamiento finalizado
```

In [ ]:
```
# Evaluar la red en el conjunto de prueba
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
```

```
        images, labels = data
        images, labels = images.to(device), labels.to(device)  # Mover datos a la G
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Precisión en el conjunto de prueba: {100 * correct / total}%')
```

```
Precisión en el conjunto de prueba: 76.7%
```