Arquitectura de LeNet-5

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

# Definir la arquitectura de LeNet-5
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(120, 84)
        self.relu4 = nn.ReLU()
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = self.relu3(self.fc1(x))
        x = self.relu4(self.fc2(x))
        x = self.fc3(x)
        return x

# Cargar el conjunto de datos MNIST y aplicar transformaciones
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Definir la red y el optimizador
net = LeNet5()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

# Entrenamiento de la red
for epoch in range(10):  # Número de épocas
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:  # Imprimir cada 100 mini lotes
            print(f'Época {epoch + 1}, Lote {i + 1}, Pérdida: {running_loss / 100:.3f}')
            running_loss = 0.0

print('Entrenamiento terminado')

# Evaluación en el conjunto de prueba
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:01<00:00, 9252487.66it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 2751281.27it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 3875625.93it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<?, ?it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Época 1, Lote 100, Pérdida: 2.271
Época 1, Lote 200, Pérdida: 0.945
Época 1, Lote 300, Pérdida: 0.289
Época 1, Lote 400, Pérdida: 0.176
Época 1, Lote 500, Pérdida: 0.160
Época 1, Lote 600, Pérdida: 0.160
Época 1, Lote 700, Pérdida: 0.126
Época 1, Lote 800, Pérdida: 0.131
Época 1, Lote 900, Pérdida: 0.091
Época 2, Lote 100, Pérdida: 0.082
Época 2, Lote 200, Pérdida: 0.071
Época 2, Lote 300, Pérdida: 0.073
Época 2, Lote 400, Pérdida: 0.072
Época 2, Lote 500, Pérdida: 0.072
Época 2, Lote 600, Pérdida: 0.055
Época 2, Lote 700, Pérdida: 0.061
Época 2, Lote 800, Pérdida: 0.070
Época 2, Lote 900, Pérdida: 0.053
Época 3, Lote 100, Pérdida: 0.045
Época 3, Lote 200, Pérdida: 0.044
Época 3, Lote 300, Pérdida: 0.050
Época 3, Lote 400, Pérdida: 0.041
Época 3, Lote 500, Pérdida: 0.052
Época 3, Lote 600, Pérdida: 0.053
Época 3, Lote 700, Pérdida: 0.045
Época 3, Lote 800, Pérdida: 0.043
Época 3, Lote 900, Pérdida: 0.044
Época 4, Lote 100, Pérdida: 0.039
Época 4, Lote 200, Pérdida: 0.036
Época 4, Lote 300, Pérdida: 0.038
Época 4, Lote 400, Pérdida: 0.038
Época 4, Lote 500, Pérdida: 0.035
Época 4, Lote 600, Pérdida: 0.034
Época 4, Lote 700, Pérdida: 0.034
Época 4, Lote 800, Pérdida: 0.036
Época 4, Lote 900, Pérdida: 0.043
Época 5, Lote 100, Pérdida: 0.027
Época 5, Lote 200, Pérdida: 0.032
Época 5, Lote 300, Pérdida: 0.032
Época 5, Lote 400, Pérdida: 0.034
Época 5, Lote 500, Pérdida: 0.026
Época 5, Lote 600, Pérdida: 0.027
Época 5, Lote 700, Pérdida: 0.029
Época 5, Lote 800, Pérdida: 0.039
Época 5, Lote 900, Pérdida: 0.034
Época 6, Lote 100, Pérdida: 0.027
Época 6, Lote 200, Pérdida: 0.025
Época 6, Lote 300, Pérdida: 0.025
Época 6, Lote 400, Pérdida: 0.024
Época 6, Lote 500, Pérdida: 0.027
Época 6, Lote 600, Pérdida: 0.024
Época 6, Lote 700, Pérdida: 0.027
Época 6, Lote 800, Pérdida: 0.027
Época 6, Lote 900, Pérdida: 0.026
Época 7, Lote 100, Pérdida: 0.025
Época 7, Lote 200, Pérdida: 0.020
Época 7, Lote 300, Pérdida: 0.018
Época 7, Lote 400, Pérdida: 0.027
Época 7, Lote 500, Pérdida: 0.020
Época 7, Lote 600, Pérdida: 0.023
Época 7, Lote 700, Pérdida: 0.023
Época 7, Lote 800, Pérdida: 0.019
Época 7, Lote 900, Pérdida: 0.023
Época 8, Lote 100, Pérdida: 0.019
Época 8, Lote 200, Pérdida: 0.015
Época 8, Lote 300, Pérdida: 0.015
Época 8, Lote 400, Pérdida: 0.018
Época 8, Lote 500, Pérdida: 0.021
Época 8, Lote 600, Pérdida: 0.016
Época 8, Lote 700, Pérdida: 0.020
Época 8, Lote 800, Pérdida: 0.026
Época 8, Lote 900, Pérdida: 0.024
Época 9, Lote 100, Pérdida: 0.020
Época 9, Lote 200, Pérdida: 0.013
Época 9, Lote 300, Pérdida: 0.015
Época 9, Lote 400, Pérdida: 0.012
Época 9, Lote 500, Pérdida: 0.011
Época 9, Lote 600, Pérdida: 0.015
Época 9, Lote 700, Pérdida: 0.017
Época 9, Lote 800, Pérdida: 0.023
Época 9, Lote 900, Pérdida: 0.024
Época 10, Lote 100, Pérdida: 0.012
Época 10, Lote 200, Pérdida: 0.013
Época 10, Lote 300, Pérdida: 0.012
Época 10, Lote 400, Pérdida: 0.011
Época 10, Lote 500, Pérdida: 0.015
Época 10, Lote 600, Pérdida: 0.023
Época 10, Lote 700, Pérdida: 0.019
Época 10, Lote 800, Pérdida: 0.020
Época 10, Lote 900, Pérdida: 0.010
Entrenamiento terminado
Exactitud en el conjunto de prueba: 98.85
```

- Métrica Utilizada: Accuracy
- Justificación: En el caso de la clasificación de dígitos escritos a mano en el conjunto de datos MNIST, la métrica accuracy fue nuestra primera opción elección. El accuracy mide la proporción de dígitos que se clasificaron correctamente en el conjunto de prueba. Dado que todas las clases tienen igual importancia, el accuracy proporciona una visión clara y fácil de interpretar del rendimiento del modelo. El objetivo es clasificar correctamente tantos dígitos como sea posible.

Arquitectura de AlexNet

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

# Definir la arquitectura de AlexNet para CIFAR-10
class AlexNetCIFAR(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNetCIFAR, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            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),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

# Cargar el conjunto de datos CIFAR-10 y aplicar transformaciones
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Definir la red y el optimizador
net = AlexNetCIFAR(num_classes=10)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=1e-4)

# Entrenamiento de la red
for epoch in range(10):  # Número de épocas
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:  # Imprimir cada 100 mini lotes
            print(f'Época {epoch + 1}, Lote {i + 1}, Pérdida: {running_loss / 100:.3f}')
            running_loss = 0.0

print('Entrenamiento terminado')

# Evaluación en el conjunto de prueba
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
files already downloaded and verified
files already downloaded and verified
Época 1, Lote 100, Pérdida: 2.303
Época 1, Lote 200, Pérdida: 2.302
Época 1, Lote 300, Pérdida: 2.299
Época 1, Lote 400, Pérdida: 2.132
Época 1, Lote 500, Pérdida: 1.934
Época 1, Lote 600, Pérdida: 1.867
Época 1, Lote 700, Pérdida: 1.867
Época 2, Lote 100, Pérdida: 1.749
Época 2, Lote 200, Pérdida: 1.733
Época 2, Lote 300, Pérdida: 1.670
Época 2, Lote 400, Pérdida: 1.628
Época 2, Lote 500, Pérdida: 1.575
Época 2, Lote 600, Pérdida: 1.575
Época 2, Lote 700, Pérdida: 1.538
Época 3, Lote 100, Pérdida: 1.427
Época 3, Lote 200, Pérdida: 1.427
Época 3, Lote 300, Pérdida: 1.400
Época 3, Lote 400, Pérdida: 1.350
Época 3, Lote 500, Pérdida: 1.348
Época 3, Lote 600, Pérdida: 1.318
Época 3, Lote 700, Pérdida: 1.283
Época 4, Lote 100, Pérdida: 1.194
Época 4, Lote 200, Pérdida: 1.257
Época 4, Lote 300, Pérdida: 1.220
Época 4, Lote 400, Pérdida: 1.318
Época 4, Lote 500, Pérdida: 1.382
Época 4, Lote 600, Pérdida: 1.319
Época 4, Lote 700, Pérdida: 1.864
Época 5, Lote 100, Pérdida: 1.022
Época 5, Lote 200, Pérdida: 1.003
Época 5, Lote 300, Pérdida: 0.995
Época 5, Lote 400, Pérdida: 1.037
Época 5, Lote 500, Pérdida: 0.993
Época 5, Lote 600, Pérdida: 0.951
Época 5, Lote 700, Pérdida: 0.965
Época 6, Lote 100, Pérdida: 0.900
Época 6, Lote 200, Pérdida: 0.867
Época 6, Lote 300, Pérdida: 0.868
Época 6, Lote 400, Pérdida: 0.865
Época 6, Lote 500, Pérdida: 0.836
Época 6, Lote 600, Pérdida: 0.831
Época 6, Lote 700, Pérdida: 0.847
Época 7, Lote 100, Pérdida: 0.791
Época 7, Lote 200, Pérdida: 0.791
Época 7, Lote 300, Pérdida: 0.777
Época 7, Lote 400, Pérdida: 0.763
Época 7, Lote 500, Pérdida: 0.744
Época 7, Lote 600, Pérdida: 0.755
Época 7, Lote 700, Pérdida: 0.765
Época 8, Lote 100, Pérdida: 0.727
Época 8, Lote 200, Pérdida: 0.708
Época 8, Lote 300, Pérdida: 0.751
Época 8, Lote 400, Pérdida: 0.681
Época 8, Lote 500, Pérdida: 0.684
Época 8, Lote 600, Pérdida: 0.675
Época 8, Lote 700, Pérdida: 0.668
Época 9, Lote 100, Pérdida: 0.652
Época 9, Lote 200, Pérdida: 0.655
Época 9, Lote 300, Pérdida: 0.639
Época 9, Lote 400, Pérdida: 0.643
Época 9, Lote 500, Pérdida: 0.643
Época 9, Lote 600, Pérdida: 0.633
Época 9, Lote 700, Pérdida: 0.629
Época 10, Lote 100, Pérdida: 0.581
Época 10, Lote 200, Pérdida: 0.563
Época 10, Lote 300, Pérdida: 0.589
Época 10, Lote 400, Pérdida: 0.619
Época 10, Lote 500, Pérdida: 0.579
Época 10, Lote 600, Pérdida: 0.589
Época 10, Lote 700, Pérdida: 0.591
Entrenamiento terminado
Exactitud en el conjunto de prueba: 78.815
```

- Métrica Utilizada: Accuracy
- Justificación: Con la clasificación de imágenes con el conjunto de datos CIFAR-10, también escogimos accuracy. La exactitud mide la proporción de imágenes que se clasificaron correctamente en el conjunto de prueba. Al igual que con el MNIST, todas las clases en CIFAR-10 tienen igual importancia, y el objetivo es lograr la mayor cantidad de clasificaciones correctas posibles. La accuracy proporciona una medida global de cuán bien se está desempeñando el modelo en la clasificación de diversas categorías de objetos en imágenes.

a. ¿Cuál es la diferencia principal entre ambas arquitecturas?

La diferencia principal entre LeNet-5 y AlexNet radica en su complejidad y profundidad. AlexNet es una red más profunda y compleja en comparación con LeNet-5. AlexNet tiene cinco capas convolucionales y tres capas completamente conectadas, mientras que LeNet-5 tiene dos capas convolucionales y tres capas completamente conectadas. También AlexNet utiliza técnicas como la normalización por lotes y dropout para regularizar y acelerar el entrenamiento.

b. ¿Podría usarse LeNet-5 para un problema como el que resolvió usando AlexNet? ¿Y viceversa?

- LeNet-5 podría utilizarse para un problema similar al resuelto con AlexNet, pero es menos probable que obtenga un rendimiento tan alto. AlexNet se diseñó para tareas de clasificación más complejas y profundas, y tiende a funcionar mejor en conjuntos de datos grandes y complicados como ImageNet.
- AlexNet si no podría usarse para un problema que LeNet-5 si maneje, especialmente si el problema es menos complejo y requiere menos capacidad de procesamiento. AlexNet es más grande y más complejo, lo que lo hace más adecuado para problemas más desafiantes.

c. Indique claramente que le pareció más interesante de cada arquitectura

- Lo que mas nos parecio interesante de LeNet-5 es su simplicidad y eficacia en la clasificación de dígitos escritos a mano en el conjunto de datos MNIST, literalmente fue un 98% de presicion. A pesar de ser una red neuronal relativamente pequeña y antigua en comparación con las arquitecturas modernas, aún puede lograr un buen rendimiento en tareas de clasificación de dígitos.
- Lo interesante de AlexNet fue su capacidad para manejar conjuntos de datos más grandes y complejos, como lo es CIFAR-10, y su capacidad para aprender características más abstractas y profundas de las imágenes.