Task 1

In [31]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score
```

Parte 1

In [32]:
```
## Se anadira droutout a la arquitectura LeNet-5 para mejorar la generalizacion del
# Definir la arquitectura LeNet-5 con Dropout
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.dropout = nn.Dropout(0.5)  # Añadir Dropout con un 50% de probabilidad

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 16*5*5)
        x = self.dropout(torch.relu(self.fc1(x)))  # Aplicar Dropout después de la
        x = self.dropout(torch.relu(self.fc2(x)))  # Aplicar Dropout después de la
        x = self.fc3(x)
        return x
```

In [33]:
```
# Cargar el dataset MNIST
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),
```

In [34]:
```
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, tra
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)
```

In [35]:
```
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, tra
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)
```

In [36]:
```
# Inicializar el modelo, la función de pérdida y el optimizador
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [37]:
```
## comprobar si se puede usar cuda con gpu.
print(device)
print(torch.cuda.is_available())
```

```
cuda
True
```

In [38]:
```
# Entrenamiento
epochs = 10
for epoch in range(epochs):
    running_loss = 0.0
    model.train()  # Asegurarse de que el modelo esté en modo de entrenamiento (Dro
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

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

        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")
```

```
Epoch 1, Loss: 0.359000263984253
Epoch 2, Loss: 0.13579924102630467
Epoch 3, Loss: 0.10573464371139804
Epoch 4, Loss: 0.08930337681182039
Epoch 5, Loss: 0.07761206855581453
Epoch 6, Loss: 0.07075405709274735
Epoch 7, Loss: 0.06550046803742492
Epoch 8, Loss: 0.06018247390315325
Epoch 9, Loss: 0.05528907021235403
Epoch 10, Loss: 0.05188241279313224
```

In [39]:
```
# Función para calcular la métrica de precisión
def calculate_accuracy(model, dataloader, device):
    model.eval()  # Poner el modelo en modo evaluación (Dropout desactivado)
    all_labels = []
    all_predictions = []

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)

            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

    accuracy = accuracy_score(all_labels, all_predictions)
    return accuracy
```

In [40]:
```
# Calcular la precisión en el conjunto de prueba
accuracy = calculate_accuracy(model, testloader, device)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
Accuracy: 98.94%
```

Parte 2

In [1]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score
```

In [2]:
```python
# Definir la arquitectura AlexNet
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, 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=3, stride=2),
        )
        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, 10),  # CIFAR10 tiene 10 clases
        )

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

In [3]:
```
# Data Augmentation y normalización para CIFAR-10
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),   # Flip horizontal aleatorio
    transforms.RandomCrop(32, padding=4), # Recorte aleatorio con padding
    transforms.Resize(224),              # Redimensionar a 224x224
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalización
])
```

In [4]:
```
# Cargar el dataset CIFAR10
trainset = torchvision.datasets.CIFAR10(root='./data2', train=True, download=True,
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

```
Files already downloaded and verified
```

In [5]:
```
testset = torchvision.datasets.CIFAR10(root='./data2', train=False, download=True,
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```

```
Files already downloaded and verified
```

In [6]:
```
# Inicializar el modelo, la función de pérdida y el optimizador
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = AlexNet().to(device)
criterion = nn.CrossEntropyLoss()
```

In [7]:
```
# Usar SGD con momentum
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
```

In [9]:
```
# Entrenamiento
epochs = 20  # Aumentar el número de épocas para mejorar el rendimiento
for epoch in range(epochs):
    running_loss = 0.0
    model.train()
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

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

        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")
```

```
Epoch 1, Loss: 1.3372866891686568
Epoch 2, Loss: 1.1041173325932545
Epoch 3, Loss: 0.9518367097048503
Epoch 4, Loss: 0.8488741616535065
Epoch 5, Loss: 0.7605342088300554
Epoch 6, Loss: 0.6945815810462093
Epoch 7, Loss: 0.6493865771747916
Epoch 8, Loss: 0.6009499154356129
Epoch 9, Loss: 0.5765223242056644
Epoch 10, Loss: 0.5356589795073585
Epoch 11, Loss: 0.5108854698441218
Epoch 12, Loss: 0.4870008224302241
Epoch 13, Loss: 0.47014428328370195
Epoch 14, Loss: 0.44887373133388625
Epoch 15, Loss: 0.431289500916553
Epoch 16, Loss: 0.41637180050087097
Epoch 17, Loss: 0.40375681627360754
Epoch 18, Loss: 0.38806021351681647
Epoch 19, Loss: 0.3780418832398132
Epoch 20, Loss: 0.36395310950667964
```

In [10]:
```python
# Función para calcular la métrica de precisión
def calculate_accuracy(model, dataloader, device):
    model.eval()
    all_labels = []
    all_predictions = []

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)

            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

    accuracy = accuracy_score(all_labels, all_predictions)
    return accuracy
```

In [11]:
```python
# Calcular la precisión en el conjunto de prueba
accuracy = calculate_accuracy(model, testloader, device)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
Accuracy: 83.02%
```

Preguntas

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

La diferencia principal entre LeNet-5 y AlexNet radica en la complejidad y el propósito:

- LeNet-5 fue diseñado para tareas sencillas como la clasificación de dígitos en imágenes pequeñas (28x28 píxeles) en el dataset MNIST. Tiene menos capas convolucionales y un número significativamente menor de parámetros.

- AlexNet es una red mucho más grande, diseñada para resolver tareas complejas de clasificación en imágenes a gran escala (como el dataset ImageNet con imágenes de 224x224 píxeles). Incluye más capas convolucionales, más filtros y utiliza técnicas como Dropout y normalización local para mejorar el rendimiento y evitar el sobreajuste.

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

- LeNet-5 en problemas como los resueltos con AlexNet: No sería adecuado, ya que LeNet-5 fue diseñado para imágenes pequeñas y problemas sencillos. Su capacidad para manejar imágenes más grandes y complejas como las del dataset CIFAR10 sería muy limitada, y probablemente su rendimiento sería muy bajo.
- AlexNet en problemas como los resueltos con LeNet-5: Aunque podría usarse, AlexNet sería innecesariamente grande y complejo para resolver problemas sencillos como la clasificación de dígitos en el dataset MNIST. Esto sería ineficiente y llevaría a un sobreajuste en este tipo de problemas simples.

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

- LeNet-5: Lo más interesante de LeNet-5 es su simplicidad y eficacia en resolver problemas de clasificación de imágenes pequeñas como MNIST. A pesar de ser una red bastante antigua, sigue siendo efectiva para tareas específicas.
- AlexNet: La introducción de técnicas como Dropout y el uso de un gran número de filtros y capas convolucionales la hace mucho más poderosa para tareas complejas. Su capacidad para procesar imágenes a gran escala y su influencia en arquitecturas modernas es lo más destacable.

Task 2

1. Investigue e indique en qué casos son útiles las siguientes arquitecturas, agregue imagenes si esto le ayuda a una mejor comprensión

   A. GoogleNet (Inception)
   GoogleNet es útil en tareas de clasificación de imágenes a gran escala donde se busca un equilibrio entre precisión y eficiencia computacional. La arquitectura Inception permite procesar múltiples escalas de características en paralelo, reduciendo el número de parámetros y evitando el sobreajuste. Es ideal para dispositivos con recursos limitados y para mejorar el rendimiento en conjuntos de datos como ImageNet.

   B. DenseNet (Densely Connected Convolutional Networks)
   DenseNet es beneficioso cuando se requiere una mejor propagación de características y gradientes a través de la red. Al conectar cada capa con todas las capas posteriores, mejora el flujo de información y reduce el problema del gradiente desaparecido. Es útil

en tareas donde se necesita una alta precisión con un número relativamente bajo de parámetros.

C. MobileNet

MobileNet es ideal para aplicaciones móviles y embebidas donde los recursos computacionales y energéticos son limitados. Utiliza convoluciones separables en profundidad para reducir significativamente el tamaño del modelo y los requisitos de cómputo, manteniendo una precisión aceptable en tareas como clasificación y detección de objetos en dispositivos con poca potencia.

D. EfficientNet

EfficientNet es útil cuando se busca maximizar la precisión mientras se minimiza el costo computacional. Introduce un método de escalamiento compuesto que equilibra la profundidad, el ancho y la resolución de la red. Es adecuado para aplicaciones que requieren alta precisión con eficiencia computacional, como servicios en la nube y aplicaciones en tiempo real.

2. ¿Cómo la arquitectura de transformers puede ser usada para image recognition?

Los transformers pueden aplicarse al reconocimiento de imágenes dividiendo una imagen en parches y tratándolos como una secuencia de tokens, similar al procesamiento en NLP. Modelos como Vision Transformer (ViT) utilizan esta técnica para capturar relaciones globales entre diferentes partes de la imagen, permitiendo un rendimiento superior en tareas de clasificación sin depender de convoluciones tradicionales.

Integrantes

- Angel Castellanos
- Diego Morales
- Alejandro Azurdia