Batch Normalization en Redes Neuronales

Objetivos de la Normalización por Lotes (Batch Normalization)

- **Acelerar el entrenamiento**: Batch normalization permite utilizar tasas de aprendizaje más altas, lo que puede acelerar el proceso de entrenamiento.
- **Reducir la sensibilidad a la inicialización de parámetros**: Estabiliza y acelera el aprendizaje ajustando las activaciones.
- **Mitigar el sobreajuste**: Actúa como una forma de regularización similar al dropout.

Cómo Funciona

- Durante el entrenamiento, para cada mini-lote, calcula la media y la desviación estándar de las activaciones.
- Normaliza las activaciones usando estas estadísticas.
- Ajusta las activaciones normalizadas usando parámetros de escala y desplazamiento aprendidos.

In [15]:
```python
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

# Transformación para normalizar los datos
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Cargar el dataset CIFAR-10
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

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

In [16]:
```python
#### Implementación de Batch Normalization en PyTorch

# Definir una arquitectura de red con batch normalization
# Definición del modelo CNN con Batch Normalization
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, 1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 64, 3, 1)
        self.bn3 = nn.BatchNorm2d(64)

        # Cálculo del tamaño de la salida después de las capas convolucionales y de pooling
        self._to_linear = None
        self.convs = nn.Sequential(
            self.conv1, self.bn1, nn.ReLU(), self.pool,
            self.conv2, self.bn2, nn.ReLU(), self.pool,
            self.conv3, self.bn3, nn.ReLU(), self.pool
        )
        self._get_conv_output((3, 32, 32))
        self.fc1 = nn.Linear(self._to_linear, 64)
        self.bn_fc1 = nn.BatchNorm1d(64)
        self.fc2 = nn.Linear(64, 10)

    def _get_conv_output(self, shape):
        o = torch.zeros(1, *shape)
        o = self.convs(o)
        self._to_linear = int(np.prod(o.size()))

    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, self._to_linear)
        x = F.relu(self.bn_fc1(self.fc1(x)))
        x = self.fc2(x)
        return x
```

In [17]:
```python
# Crear una instancia de la red
net = Net()
net
```

Out[17]:
```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
  (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (convs): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=256, out_features=64, bias=True)
  (bn_fc1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

In [18]:
```
# Definir la función de pérdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Variables para almacenar la pérdida y la precisión
train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []
```

In [19]:
```
# Función para calcular la precisión
def calculate_accuracy(loader, model):
    correct = 0
    total = 0
    with torch.no_grad():
        for data in loader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

In [21]:
```python
# Entrenamiento del modelo
for epoch in range(2):  # Loop de entrenamiento
    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()

    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)

    test_loss = 0.0
    for data in testloader:
        images, labels = data
        outputs = net(images)
        loss = criterion(outputs, labels)
        test_loss += loss.item()

    test_loss /= len(testloader)
    test_losses.append(test_loss)

    train_accuracy = calculate_accuracy(trainloader, net)
    train_accuracies.append(train_accuracy)

    test_accuracy = calculate_accuracy(testloader, net)
    test_accuracies.append(test_accuracy)

    print(f'Epoch {epoch+1}, Train Loss: {train_loss:.3f}, Test Loss: {test_loss:.3f}, Train Accuracy: {train

print('Finished Training')
```

```
Epoch 1, Train Loss: 1.353, Test Loss: 1.314, Train Accuracy: 57.13, Test Accuracy: 54.77
Epoch 2, Train Loss: 1.250, Test Loss: 1.233, Train Accuracy: 60.48, Test Accuracy: 58.37
Finished Training
```

In [ ]:
```
# Visualización de las curvas de pérdida y precisión
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Curves')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy Curves')
plt.legend()
plt.show()
```

In [ ]:

In [ ]:

Explicación del Código

1. **Definición de la Red con Batch Normalization**:

   - Se añaden capas de batch normalization (`nn.BatchNorm2d` para convoluciones y `nn.BatchNorm1d` para capas totalmente conectadas).

2. **Entrenamiento del Modelo**:

   - Se entrena el modelo utilizando CIFAR-10, mostrando la pérdida en cada época.

Batch normalization ayuda a estabilizar y acelerar el entrenamiento de redes neuronales profundas, haciendo que el proceso sea más robusto y eficiente.

In [ ]:

```
```