Implementation of LeNet using PyTorch

Prepared by :

SRIHARI MADDINENI

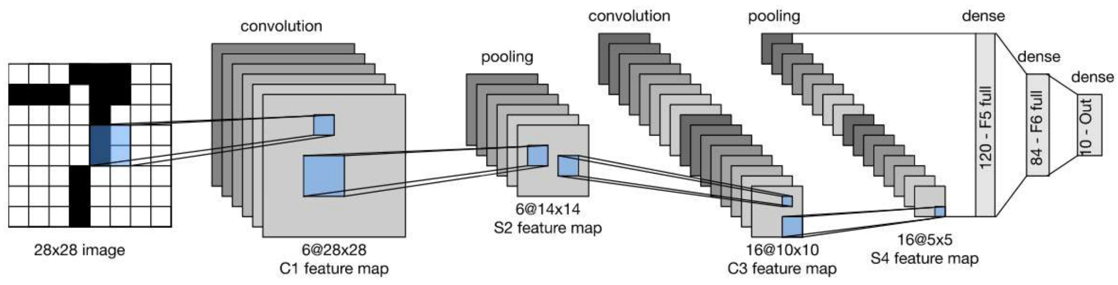
Y20AIT507

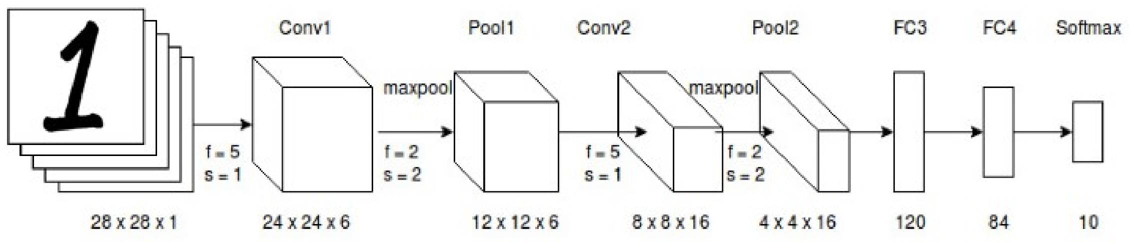
Introduction to LeNet:

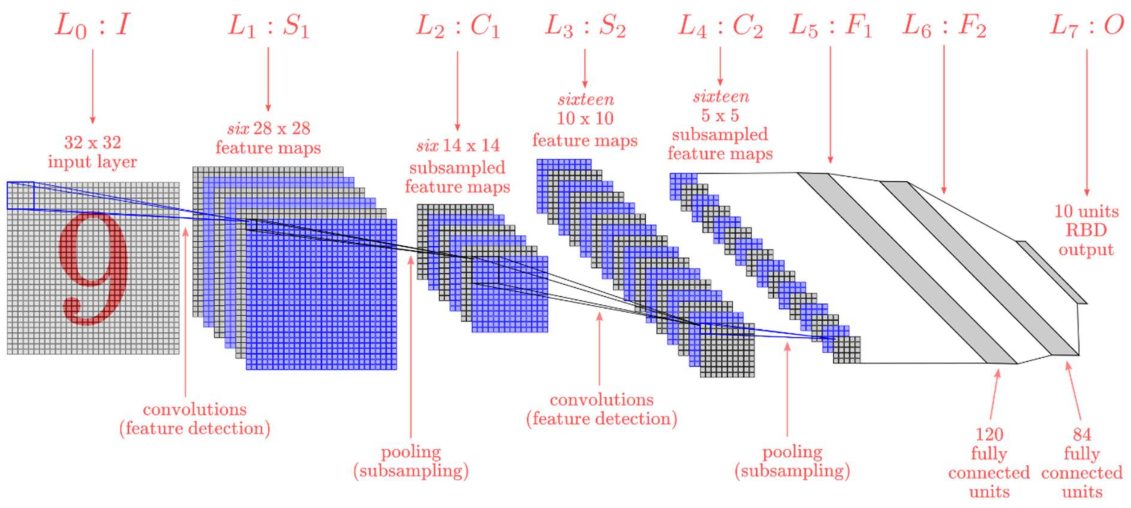
- LeNet-5 is a classic convolutional neural network architecture developed by Yann LeCun et al. in 1998.
- It was designed for handwritten digit recognition tasks.
- Despite its simplicity compared to modern architectures, LeNet laid the foundation for deep learning in computer vision.

PyTorch Overview:

- PyTorch is a popular open-source machine learning library developed by Facebook AI.
- It provides tensor computation with strong GPU acceleration and deep neural networks built on a dynamic computation graph.
- PyTorch is widely used for research and production in various domains, including computer vision.






$L_0 : I$
$L_1 : S_1$
$L_2 : C_1$
$L_3 : S_2$
$L_4 : C_2$
$L_5 : F_1$
$L_6 : F_2$
$L_7 : O$
32 x 32 input layer
six 28 x 28 feature maps
six 14 x 14 subsampled feature maps
sixteen 10 x 10 feature maps
sixteen 5 x 5 subsampled feature maps
10 units RBD output
convolutions (feature detection)
pooling (subsampling)
convolutions (feature detection)
pooling (subsampling)
120 fully connected units
84 fully connected units

Implementation of LeNet using PyTorch:

```
#Step 1: Importing necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms


#Step 2: Define the LeNet architecture

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)


    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = torch.nn.functional.max_pool2d(x, 2)
        x = torch.nn.functional.relu(self.conv2(x))
        x = torch.nn.functional.max_pool2d(x, 2)
        x = x.view(-1, 16 * 5 * 5)
        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.nn.functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x


#Step 3: Prepare the data

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=32, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)


#Step 4: Define loss function and optimizer
net = LeNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)


#Step 5: Train the network

for epoch in range(5):  # loop over the dataset multiple times
    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:    # print every 100 mini-batches
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0
print('Finished Training')
```

```
#Step 6: Evaluate the network

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct / total))
```

Implementing LeNet using PyTorch for handwritten classification involves several key steps:

1. **Architecture Design**: LeNet, comprising convolutional layers, subsampling layers (max-pooling), and fully connected layers, is designed within PyTorch. Each layer's architecture, including the number of filters, kernel sizes, and activation functions, is specified.
2. **PyTorch Modules**: PyTorch's nn.Module class is utilized to define the LeNet architecture. This class allows for modular construction, encapsulating layers and operations within separate modules for organization and ease of implementation.
3. **Data Preparation**: Handwritten digit images, such as those from the MNIST dataset, are preprocessed and loaded into PyTorch's DataLoader. Data augmentation techniques may be applied to enhance model generalization.
4. **Training Process**: The LeNet model is trained using PyTorch's automatic differentiation capabilities facilitated by the autograd module. Gradient descent optimization algorithms, such as stochastic gradient descent (SGD) or Adam, are employed to minimize the loss function and update the model parameters iteratively.
5. **Evaluation**: The trained LeNet model is evaluated using a separate validation dataset to assess its performance. Metrics such as accuracy, precision, recall, and F1-score are computed to quantify the model's effectiveness in handwritten digit classification.
6. **Testing**: Finally, the model is tested on unseen handwritten digit images to evaluate its real-world performance. Test accuracy and other relevant metrics are computed to validate the model's robustness and generalization capability.

By leveraging PyTorch's functionalities, including its modular design, automatic differentiation, and optimization capabilities, implementing LeNet for handwritten classification becomes efficient and effective. PyTorch streamlines the development process, allowing researchers and practitioners to focus on model design and experimentation, ultimately leading to accurate and reliable handwritten digit recognition systems.

Interpretation of example.

In this example, we are implementing LeNet, a convolutional neural network architecture, for the task of handwritten digit recognition using the MNIST dataset. Handwritten digit recognition is a real-life problem that has numerous applications, including:

1. Postal Automation: Automated sorting of mail by reading handwritten zip codes or addresses.
2. Banking: Recognizing handwritten digits on checks for automatic processing.
3. Captcha Recognition: Identifying handwritten characters in CAPTCHA challenges on websites.
4. OCR (Optical Character Recognition): Converting handwritten documents into digital text.
5. Form Processing: Automatically extracting information from handwritten forms, such as surveys or application forms.

The MNIST dataset consists of 28x28 grayscale images of handwritten digits (0-9) and corresponding labels. By training a neural network like LeNet on this dataset, we aim to develop a model that can accurately classify and recognize handwritten digits.

The practical application of this example lies in building systems that can automate tasks involving handwritten digit recognition, leading to increased efficiency and accuracy in various domains such as postal services, banking, document processing, and security.

Conclusion:

In this implementation, we learned how to create and train LeNet using PyTorch.

PyTorch provides a flexible and intuitive framework for building and training neural networks.

References:

1. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.
2. PyTorch Documentation. Retrieved from https://pytorch.org/docs/stable/index.htm
3. Official PyTorch GitHub Repository. Retrieved from https://github.com/pytorch/pytorch
4. Wikipedia contributors. (2022). MNIST database. In Wikipedia, The Free Encyclopedia. Retrieved January 30, 2024, from https://en.wikipedia.org/wiki/MNIST_database

GUIDED BY:

DR.SIVARAM NALLURI

HOD, IT

BEC, BAPATLA

Prepared By :

SRIHARI MADDINENI

Y20AIT507