Aim: PyTorch implementation of CNN

Software requirement for Python: Jupyter Notebook

Theory:

To implement a Convolutional Neural Network (CNN) using PyTorch, you can follow these general steps:

1. Import the necessary modules and packages, including torch, torchvision.datasets, torchvision.transforms, torch.nn, and torch.optim.
2. Define the hyperparameters for your model, including the number of epochs, batch size, and learning rate.
3. Define the architecture of your CNN using PyTorch's nn.Module class. This involves defining the various layers of your CNN, including convolutional layers, pooling layers, and fully connected layers.
4. Load your dataset and preprocess the data using the transforms module from torchvision. This includes transformations such as converting images to tensors and normalizing pixel values.
5. Create data loaders for your training and test datasets using torch.utils.data.DataLoader.
6. Instantiate your CNN model and define the loss function and optimizer.
7. Train your CNN by iterating over the training dataset, computing forward and backward passes, and optimizing the model weights using the optimizer.
8. Evaluate the performance of your model on the test dataset by computing predictions and comparing them to the true labels.
9. Save the trained model parameters using torch. save for future use.

Procedure:

The code is implementing a convolutional neural network (CNN) using PyTorch to perform image classification on the MNIST dataset.

First, the code defines hyperparameters such as the number of epochs, batch size, and learning rate. Then, it defines the CNN architecture as a class with several convolutional layers, activation functions, and a fully connected layer.

The MNIST dataset is loaded and transformed using PyTorch's built-in functions, and then the data is split into training and test sets using data loaders.

The CNN model is instantiated, and the loss function (cross-entropy) and optimizer (Adam) are defined. The model is then trained for the specified number of epochs, with the loss being printed every 100 steps.

After training, the model is tested on the test set to calculate its accuracy. Finally, the trained model is saved to a file using PyTorch's state_dict() function.

Conclusion: Thus, we have studied PyTorch implementation of CNN

Program Code and Output

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision.datasets as datasets

import torchvision.transforms as transforms


# Define hyperparameters

num_epochs = 5

batch_size = 64

learning_rate = 0.001


# Define the CNN architecture

class CNN(nn.Module):

    def __init__(self):

        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)

        self.relu1 = nn.ReLU()

        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)

        self.relu2 = nn.ReLU()

        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(in_features=7*7*32, out_features=10)


    def forward(self, x):

        x = self.conv1(x)

        x = self.relu1(x)

        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu2(x)

        x = self.pool2(x)

        x = x.view(-1, 7*7*32)

        x = self.fc1(x)

        return x


# Load the MNIST dataset

train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(),
download=True)

test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())


# Create the dataloaders

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)


# Instantiate the CNN model and define the loss function and optimizer

model = CNN()

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning_rate)


# Train the model

for epoch in range(num_epochs):

    for i, (images, labels) in enumerate(train_loader):

        # Forward pass

        outputs = model(images)

        loss = criterion(outputs, labels)


        # Backward and optimize

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                  .format(epoch+1, num_epochs, i+1, len(train_loader), loss.item()))

# Test the model
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
# Save the trained model
torch.save(model.state_dict(), 'cnn.ckpt')
```

OUTPUT OF THE CODE

```
>>> %Run -c $EDITOR_CONTENT

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz

Processing

Done!

C:\Users\ted7\AppData\Roaming\Python\Python310\site-packages\torchvision\transforms.py:36:
UserWarning: TypedStorage is deprecated. It will be removed in the future and UntypedStorage will be
the only storage class. This should only matter to you if you are using storages directly.  To access
UntypedStorage directly, use tensor.untyped_storage() instead of tensor.storage()

  img = torch.ByteTensor(torch.ByteStorage.from_buffer(pic.tobytes()))

Epoch [1/5], Step [100/938], Loss: 0.2831

Epoch [1/5], Step [200/938], Loss: 0.3648

Epoch [1/5], Step [300/938], Loss: 0.1767

Epoch [1/5], Step [400/938], Loss: 0.1572

Epoch [1/5], Step [500/938], Loss: 0.1258

Epoch [1/5], Step [600/938], Loss: 0.1229

Epoch [1/5], Step [700/938], Loss: 0.0968

Epoch [1/5], Step [800/938], Loss: 0.0415

Epoch [1/5], Step [900/938], Loss: 0.0414

Epoch [2/5], Step [100/938], Loss: 0.0179

Epoch [2/5], Step [200/938], Loss: 0.0380

Epoch [2/5], Step [300/938], Loss: 0.0672

Epoch [2/5], Step [400/938], Loss: 0.1538

Epoch [2/5], Step [500/938], Loss: 0.0923

Epoch [2/5], Step [600/938], Loss: 0.1063

Epoch [2/5], Step [700/938], Loss: 0.1835

Epoch [2/5], Step [800/938], Loss: 0.0842
Epoch [2/5], Step [900/938], Loss: 0.0146

Epoch [3/5], Step [100/938], Loss: 0.0270

Epoch [3/5], Step [200/938], Loss: 0.1467

Epoch [3/5], Step [300/938], Loss: 0.0634

Epoch [3/5], Step [400/938], Loss: 0.0171

Epoch [3/5], Step [500/938], Loss: 0.0640

Epoch [3/5], Step [600/938], Loss: 0.0037

Epoch [3/5], Step [700/938], Loss: 0.0324

Epoch [3/5], Step [800/938], Loss: 0.0483

Epoch [3/5], Step [900/938], Loss: 0.0074

Epoch [4/5], Step [100/938], Loss: 0.0138

Epoch [4/5], Step [200/938], Loss: 0.0166

Epoch [4/5], Step [300/938], Loss: 0.0386

Epoch [4/5], Step [400/938], Loss: 0.0541

Epoch [4/5], Step [500/938], Loss: 0.0296

Epoch [4/5], Step [600/938], Loss: 0.0387

Epoch [4/5], Step [700/938], Loss: 0.0138

Epoch [4/5], Step [800/938], Loss: 0.0690

Epoch [4/5], Step [900/938], Loss: 0.0088

Epoch [5/5], Step [100/938], Loss: 0.0230

Epoch [5/5], Step [200/938], Loss: 0.0729

Epoch [5/5], Step [300/938], Loss: 0.0081

Epoch [5/5], Step [400/938], Loss: 0.0435

Epoch [5/5], Step [500/938], Loss: 0.0470

Epoch [5/5], Step [600/938], Loss: 0.0420

Epoch [5/5], Step [700/938], Loss: 0.0339

Epoch [5/5], Step [800/938], Loss: 0.0064

Epoch [5/5], Step [900/938], Loss: 0.0098

Test Accuracy of the model on the 10000 test images: 98.75 %
```