MLAI 504
NEURAL NETWORKS & DEEP LEARNING

MLAI – 2023/2024

Dr. Zein Al Abidin IBRAHIM

zein.ibrahim@ul.edu.lb

IMPLEMENTATION OF CNN USING PYTORCH & KERAS

NEURAL NETWORK

EXAMPLE

- Importing libraries

```
# Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

EXAMPLE

- Defining the architecture of the network

```
# Define the CNN architecture
# Output size= ((Input size - Kernel size + 2×Padding)/Stride) +1
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        #output here = (32 - 16 + 2)/1 + 1 -> 3*10*10
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(32 * 16 * 16, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 16 * 16)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

EXAMPLE

- Defining the architecture of the network

1. **First Convolutional Layer (`self.conv1`):**
   - **`nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)`**
     - Input channels: 3 (assuming RGB input)
     - Output channels: 16
     - Kernel size: 3×3
     - Stride: 1
     - Padding: 1
     - Calculation for output size:

$$\text{output\_size} = \left\lfloor \frac{\text{input\_size} - \text{kernel\_size} + 2 \times \text{padding}}{\text{stride}} \right\rfloor + 1$$

$$\text{output\_size} = \left\lfloor \frac{32 - 3 + 2 \times 1}{1} \right\rfloor + 1 = 32$$

     - Output size: $16 \times 32 \times 32$ (16 channels, each of size 32×32)

EXAMPLE

- Defining the architecture of the network

2. **Second Convolutional Layer (`self.conv2`):**
   - `nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)`
     - Input channels: 16 (output channels from the previous layer)
     - Output channels: 32
     - Kernel size: 3×3
     - Stride: 1
     - Padding: 1
     - Calculation for output size:

$$\text{output\_size} = \left\lfloor \frac{\text{input\_size} - \text{kernel\_size} + 2 \times \text{padding}}{\text{stride}} \right\rfloor + 1$$

$$\text{output\_size} = \left\lfloor \frac{32 - 3 + 2 \times 1}{1} \right\rfloor + 1 = 32$$

     - Output size: $32 \times 32 \times 32$ (32 channels, each of size 32×32)

EXAMPLE

- Defining the architecture of the network

3. **Max Pooling Layer (`self.pool`):**
   - `nn.MaxPool2d(kernel_size=2, stride=2, padding=0)`
     - Kernel size: 2×2
     - Stride: 2
     - Padding: 0
     - Max pooling reduces the spatial dimensions by a factor of the kernel size. Therefore, after this layer, the output size becomes half in both height and width.
     - Output size: $32 \div 2 \times 32 \div 2 \times 32 = 16 \times 16 \times 32$ (32 channels, each of size 16×16)

EXAMPLE

- Defining the architecture of the network

4. **First Fully Connected Layer (`self.fc1`):**
   - `nn.Linear(32 * 16 * 16, 128)`
     - Input features: $32 \times 16 \times 16$ (flattened from the output of the previous layer)
     - Output features: 128
5. **Second Fully Connected Layer (`self.fc2`):**
   - `nn.Linear(128, 10)`
     - Input features: 128 (output features from the previous layer)
     - Output features: 10 (assuming this is a classification task with 10 classes)

EXAMPLE

- Load dataset and create the network

```
# Set random seed for reproducibility
torch.manual_seed(42)

# Load CIFAR-10 dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Initialize the model, loss function, and optimizer
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

EXAMPLE

- Load dataset and create the network

1. **Data Transformations:**
   - `transforms.Compose`: Composes several image transformations together.
   - `transforms.ToTensor()`: Converts the image data to PyTorch tensors.
   - `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Normalizes the tensor values to have a mean of 0.5 and a standard deviation of 0.5 for each channel. This helps in stabilizing and speeding up the training process.
2. **Load CIFAR-10 Dataset:**
   - `torchvision.datasets.CIFAR10`: Loads the CIFAR-10 dataset.
   - `root='./data'`: Specifies the directory where the dataset will be downloaded.
   - `train=True` and `train=False`: Indicates whether to load the training set or the test set.
   - `download=True`: Downloads the dataset if not already present.
   - `transform=transform`: Applies the specified transformations to the loaded data.

EXAMPLE

- Load dataset and create the network

3. **Data Loaders:**
   - `DataLoader`: PyTorch utility for batching and shuffling the data during training.
   - `train_loader`: DataLoader for the training set.
     - `batch_size=64`: Sets the batch size to 64, meaning each iteration during training will process 64 images at a time.
     - `shuffle=True`: Shuffles the training data before each epoch to introduce randomness and prevent the model from learning the order of the data.
   - `test_loader`: DataLoader for the test set with similar batch size but no shuffling.

EXAMPLE

- Training

```
# Training the CNN
num_epochs = 5

for epoch in range(num_epochs):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

EXAMPLE

- Training

```
# Training the CNN
num_epochs = 5

for epoch in range(num_epochs):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

1. **Outer Loop (`for epoch in range(num_epochs):`):**
   - This loop iterates over the specified number of epochs (`num_epochs`), representing the number of times the entire training dataset is passed through the neural network.
2. **Inner Loop (`for images, labels in train_loader:`):**
   - This loop iterates over batches of data from the training DataLoader (`train_loader`). The DataLoader is responsible for providing batches of input images (`images`) and their corresponding labels (`labels`) during each iteration.

EXAMPLE

- Training

```
# Training the CNN
num_epochs = 5

for epoch in range(num_epochs):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

3. **Zeroing Gradients (`optimizer.zero_grad()`):**
   - Before computing the gradients, the `zero_grad()` method is called on the optimizer. This is necessary because PyTorch accumulates gradients on subsequent backward passes, and calling `zero_grad()` ensures that the gradients from the previous iteration are cleared.
4. **Forward Pass (`outputs = model(images)`):**
   - The input images are passed through the neural network (`model`) to obtain the predicted outputs (`outputs`). This is the forward pass.

EXAMPLE

```python
# Training the CNN
num_epochs = 5

for epoch in range(num_epochs):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

5. **Loss Computation (`loss = criterion(outputs, labels)`):**
   - The predicted outputs are compared to the ground truth labels using a specified loss function (`criterion`). The result is the loss, which quantifies how well the model is performing on the current batch.
6. **Backward Pass (`loss.backward()`):**
   - Gradients with respect to the model parameters are computed by calling `backward()` on the loss. This is the backward pass.
7. **Gradient Descent Update (`optimizer.step()`):**
   - The optimizer (`optimizer`) updates the model parameters based on the computed gradients. This step is crucial for training the model and improving its performance.
8. **Print Training Information (`print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')`):**
   - After completing one pass through the entire training dataset (one epoch), the code prints the average loss for that epoch. The `loss.item()` retrieves the scalar value of the loss.

EXAMPLE

- Testing

```
# Testing the CNN
model.eval()
correct, total = 0, 0

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

accuracy = correct / total
print(f'Test Accuracy: {100 * accuracy:.2f}%')
```

EXAMPLE

- Testing

```
# Testing the CNN
model.eval()
correct, total = 0, 0

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

accuracy = correct / total
print(f'Test Accuracy: {100 * accuracy:.2f}%')
```

1. **Set Model to Evaluation Mode (`model.eval()`):**
   - The model is set to evaluation mode using **`model.eval()`**. This is important because it informs the model that it is being evaluated, not trained. Some layers, like dropout layers, may behave differently during training and evaluation. Setting the model to evaluation mode ensures consistent behavior during testing.
2. **Variables for Correct and Total Predictions (`correct, total = 0, 0`):**
   - Two variables, **`correct`** and **`total`**, are initialized to zero. These will be used to track the number of correctly predicted samples and the total number of test samples, respectively.
3. **Disable Gradient Computation (`with torch.no_grad():`):**
   - Inside a **`with torch.no_grad():`** block, gradient computation is disabled. This is done to save memory and computation time during testing since gradients are not needed for parameter updates during this phase.

EXAMPLE

- Testing

```
# Testing the CNN
model.eval()
correct, total = 0, 0

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

accuracy = correct / total
print(f'Test Accuracy: {100 * accuracy:.2f}%')
```

4. **Loop Over Batches in the Test DataLoader (`for images, labels in test_loader`):**
   - The code then iterates over batches of data from the test DataLoader (`test_loader`), providing test images (`images`) and their corresponding labels (`labels`).
5. **Forward Pass (`outputs = model(images)`):**
   - The test images are passed through the trained model (`model`) to obtain predicted outputs (`outputs`). These outputs typically represent the model's confidence scores or probabilities for each class.
6. **Extract Predicted Class Indices (`_, predicted = torch.max(outputs.data, 1)`):**
   - The `torch.max` function is used to extract the index of the maximum value along the specified dimension (in this case, dimension 1, which corresponds to the class dimension). The resulting `predicted` tensor contains the predicted class indices.

EXAMPLE

- Testing

```python
# Testing the CNN
model.eval()
correct, total = 0, 0

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

accuracy = correct / total
print(f'Test Accuracy: {100 * accuracy:.2f}%')
```

7. **Update Total Number of Test Samples (`total += labels.size(0)`):**
   - The total number of test samples is updated by adding the size of the current batch (`labels.size(0)`).
8. **Update Number of Correctly Predicted Samples (`correct += (predicted == labels).sum().item()`):**
   - The number of correctly predicted samples is updated by counting how many predictions match the true labels (`(predicted == labels).sum().item()`) and adding that count to the `correct` variable.