جامعة نيويورك أبوظبي

NYU | ABU DHABI

Applied Machine Learning - Spring 2025 - Lab6

Asini Subanya ash9840

The CNN model implemented in this lab, designed for image classification, specifically for the CIFAR-10 dataset is based on the GoogLeNet/Inception architecture, including the location of each module in the code. The model is implemented in PyTorch and is based on the Inception architecture.

The CNN model has following primary components:

1. **Data Loading and Preprocessing**: Uses `torchvision.datasets` and `torch.utils.data.DataLoader` for handling the dataset.
2. **Inception Module**: A custom module that implements the core Inception block with parallel convolutional pathways.
3. **CNN Model**: The main model architecture that stacks an initial convolutional layer and multiple Inception blocks followed by pooling and fully connected layers.
4. **Loss Function and Optimizer**: Uses `nn.CrossEntropyLoss` as the loss function and `torch.optim.Adam` as the optimizer.
5. **Training Loop**: Iterates through the dataset, performs forward and backward passes, and updates model parameters.

a. Data Loading and Preprocessing

.
This module serves as the entry point for the dataset, employing torchvision.datasets.CIFAR10 to automatically download and organize the CIFAR-10 data into training and testing sets. It makes use of torchvision.transforms.Compose to define a pipeline of image transformations, including random horizontal flips for training data augmentation, and conversion to PyTorch tensors, ensuring that the images are formatted appropriately for the model. Finally, torch.utils.data.DataLoader creates data loaders to manage batching and shuffling of the data, optimizing memory usage and introducing stochasticity during training.

```
train_dataset = dset.CIFAR10(root='./data', train=True, transform=train_transform,
download=True)
test_dataset = dset.CIFAR10(root='./data', train=False, transform=test_transform,
download=True)

# Data loaders
train_loader = data.DataLoader(dataset=train_dataset, batch_size=batch_train_size,
shuffle=True)
test_loader = data.DataLoader(dataset=test_dataset, batch_size=batch_test_size,
shuffle=False)
```

- **Modules**:
  - `torchvision.datasets.CIFAR10`: Used to load the CIFAR-10 dataset.
  - `torchvision.transforms.Compose`: Used to define image transformations.
  - `torchvision.transforms.RandomHorizontalFlip`: Used for data augmentation (horizontal flipping).
  - `torchvision.transforms.ToTensor`: Used to convert images to tensors.
  - `torch.utils.data.DataLoader`: Used to create data loaders for batching and shuffling the data.
  - The `train_dataset` and `test_dataset` are created using `dset.CIFAR10`, specifying the root directory, train/test split, transformations, and download option.
  - `train_transform` includes random horizontal flips for data augmentation and conversion to tensors.
  - `test_transform` only includes conversion to tensors.
  - `train_loader` and `test_loader` are created using `data.DataLoader`, which provides shuffled batches of training data and batches of test data.

b. Inception Module

The Inception module is the main building block of the CNN, designed to extract features at multiple scales through parallel convolutional pathways. Within this module, input feature maps are processed simultaneously by four branches: a 1x1 convolution for dimensionality reduction, a 1x1 convolution followed by a 3x3 convolution to capture local patterns, a 1x1 convolution followed by a 5x5 convolution to capture broader contextual information, and a 3x3 max-pooling layer followed by a 1x1 convolution, providing a shortcut for preserving spatial information. The outputs from each branch are then combined along the channel dimension, allowing the module to capture a diverse set of features at various resolutions.

```
class Inception(nn.Module):
    def __init__(self, in_planes, kernel_1_x, kernel_3_in, kernel_3_x, kernel_5_in, kernel_5_x,
pool_planes):
        super(Inception, self).__init__()

        # 1x1 conv branch
        self.b1 = nn.Sequential(
            nn.Conv2d(in_planes, kernel_1_x, kernel_size=1),
            nn.BatchNorm2d(kernel_1_x),
            nn.ReLU(True),
        )

        # 1x1 conv -> 3x3 conv branch
        self.b2 = nn.Sequential(
            nn.Conv2d(in_planes, kernel_3_in, kernel_size=1),
```

```
            nn.BatchNorm2d(kernel_3_in),
            nn.ReLU(True),
            nn.Conv2d(kernel_3_in, kernel_3_x, kernel_size=3, padding=1),
            nn.BatchNorm2d(kernel_3_x),
            nn.ReLU(True),
        )

        # 1x1 conv -> 5x5 conv branch
        self.b3 = nn.Sequential(
            nn.Conv2d(in_planes, kernel_5_in, kernel_size=1),
            nn.BatchNorm2d(kernel_5_in),
            nn.ReLU(True),
            nn.Conv2d(kernel_5_in, kernel_5_x, kernel_size=5, padding=2),
            nn.BatchNorm2d(kernel_5_x),
            nn.ReLU(True),
        )

        # 3x3 max pool -> 1x1 conv branch
        self.b4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            nn.Conv2d(in_planes, pool_planes, kernel_size=1),
            nn.BatchNorm2d(pool_planes),
            nn.ReLU(True),
        )

    def forward(self, x):
        out1 = self.b1(x)
        out2 = self.b2(x)
        out3 = self.b3(x)
        out4 = self.b4(x)
        return torch.cat([out1, out2, out3, out4], 1)
```

- `nn.Conv2d`: Convolutional layers for different branches.
- `nn.BatchNorm2d`: Batch normalization layers.
- `nn.ReLU`: ReLU activation functions.
- `nn.MaxPool2d`: Max pooling layer in one of the branches.
- The `Inception` module performs multiple parallel convolutions on the input feature maps. It has four branches:
    1. 1x1 convolution.
    2. 1x1 convolution followed by 3x3 convolution.
    3. 1x1 convolution followed by 5x5 convolution.
    4. 3x3 max pooling followed by 1x1 convolution.
- The outputs from all four branches are combined along the channel dimension.

c. CNN Model

The CNN model orchestrates the overall architecture, creating a network composed of an initial convolutional layer followed by a sequence of Inception modules and pooling layers. It begins with a Conv2D layer to learn initial features from the input images, succeeded by batch normalization and ReLU activation to stabilize training and introduce non-linearity. Subsequent Inception blocks, configured with specific channel dimensions as indicated in the architecture diagram in the lab manual, are interspersed with max-pooling layers to downsample the feature maps and progressively extract higher-level features. Global average pooling is applied to reduce spatial dimensions, and a final fully connected layer maps the learned features to the output classes, producing classification scores for the CIFAR-10 dataset.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # Initial Conv layer
        self.conv1 = nn.Conv2d(3, 192, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(192)
        self.relu1 = nn.ReLU(inplace=True)

        # First group of Inception blocks
        self.inception1a = Inception(192, 64, 96, 128, 16, 32, 32)  # outputs 256 channels
        self.inception1b = Inception(256, 128, 128, 192, 32, 96, 64)  # outputs 480 channels
        self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Second group of Inception blocks
        self.inception2a = Inception(480, 192, 96, 208, 16, 48, 64)  # outputs 512 channels
        self.inception2b = Inception(512, 160, 112, 224, 24, 64, 64)  # outputs 512 channels
        self.inception2c = Inception(512, 128, 128, 256, 24, 64, 64)  # outputs 512 channels
        self.inception2d = Inception(512, 112, 144, 288, 32, 64, 64)  # outputs 528 channels
        self.inception2e = Inception(528, 256, 160, 320, 32, 128, 128)  # outputs 832
channels
        self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Third group of Inception blocks
        self.inception3a = Inception(832, 256, 160, 320, 32, 128, 128)  # outputs 832
channels
        self.inception3b = Inception(832, 384, 192, 384, 48, 128, 128)  # outputs 1024
channels

        # Final layers
        self.avgpool = nn.AvgPool2d(kernel_size=8)  # Global average pooling (8x8 feature
maps)
        self.fc = nn.Linear(1024, 10)  # Final fully connected layer

    def forward(self, x):
        # Initial layer
```

```
        x = self.relu1(self.bn1(self.conv1(x)))

        # First group
        x = self.inception1a(x)
        x = self.inception1b(x)
        x = self.maxpool1(x)

        # Second group
        x = self.inception2a(x)
        x = self.inception2b(x)
        x = self.inception2c(x)
        x = self.inception2d(x)
        x = self.inception2e(x)
        x = self.maxpool2(x)

        # Third group
        x = self.inception3a(x)
        x = self.inception3b(x)

        # Final layers
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x


model = CNN().to(device)
```

1. `nn.Conv2d`: Initial convolutional layer.
2. `nn.BatchNorm2d`: Batch normalization layer after the initial convolution.
3. `nn.ReLU`: ReLU activation function.
4. `nn.MaxPool2d`: Max pooling layers.
5. `Inception`: Instances of the `Inception` module.
6. `nn.Linear`: Fully connected layer for classification
7. The network starts with a `Conv2D` layer with a 3x3 kernel, followed by `BatchNorm2D` and `ReLU`.
8. This is followed by a `MaxPool2D` layer.
9. An `Inception` block is applied.
10. Another `MaxPool2D` layer follows.
11. Finally, the output is passed to a fully connected (`Linear`) layer that maps the flattened features to the 10 output classes of CIFAR-10.
12. The forward pass consists of passing the input through the initial convolutional block, the Inception module, and the final fully connected layer.

d. Loss Function and Optimizer

This module does the learning process of the CNN, employing torch.optim.Adam to update the model's parameters based on the computed gradients, with the learning rate and weight decay configured to control the speed and regularization of training. A multi-step learning rate scheduler adjusts the learning rate during training to promote convergence and prevent overshooting. The nn.CrossEntropyLoss function measures the discrepancy between the predicted class probabilities and the true labels, optimizing the process towards minimizing classification errors.

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-4)

scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 100],
gamma=0.5)

criterion = nn.CrossEntropyLoss()
```

- `torch.optim.Adam`: Adam optimizer for updating the model's parameters.
- `torch.optim.lr_scheduler.MultiStepLR`: Learning rate scheduler.
- `nn.CrossEntropyLoss`: Cross-entropy loss function for classification.

- The Adam optimizer is used with a learning rate of 0.001 and weight decay of 1e-4.
- A multi-step learning rate scheduler is used to reduce the learning rate at milestones 50 and 100.
- The cross-entropy loss is used to measure the difference between the predicted and true labels.

e. Training Loop

The training loop iteratively refines the CNN model by processing the dataset in batches, performing forward and backward passes, and updating model parameters. In each epoch, the loop feeds batches of training images through the model, calculates the loss using the nn.CrossEntropyLoss, computes gradients using loss.backward(), and updates the model's weights using optimizer.step(). The loop also tracks the training and test accuracy after each epoch, providing insights into the model's generalization performance and convergence.

```
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
```

```
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

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

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    train_accuracy = 100. * correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    # Step the scheduler
    scheduler.step()

    # Evaluate
    model.eval()
    test_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    test_accuracy = 100. * correct / total
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    print(f"Epoch [{epoch+1}/{num_epochs}]\n"
          f"Train Accuracy: {train_accuracy:.2f}% | Train Loss: {train_loss:.4f}\n"
          f"Test Accuracy: {test_accuracy:.2f}% | Test Loss: {test_loss:.4f}")

# Save model
torch.save(model.state_dict(), 'cnn_cifar10.pth')

# Plot loss
plt.figure()
```

```
plt.plot(range(num_epochs), train_losses, label='Train Loss')
plt.plot(range(num_epochs), test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss over Epochs')
plt.show()
```

- The training loop iterates through the specified number of epochs.
- In each epoch, the training data is processed in batches.
- For each batch:

    1. The model makes predictions on the input data.
    2. The loss is calculated using the cross-entropy loss function.
    3. The gradients are calculated using `loss.backward()`.
    4. The model parameters are updated using `optimizer.step()`.

- The test data is used to evaluate the model's performance after each epoch.
- Training and test accuracy and loss are printed for each epoch.