CNN Data Transformation and Training - PyTorch

Akarshan Ghosh

April 2025

1 Introduction

This document provides a comprehensive overview of using PyTorch for image classification with the CIFAR-10 dataset. It covers data transformations, dataset loading, Convolutional Neural Network (CNN) model definition, and model training and saving. Each section includes code snippets and detailed explanations to facilitate understanding of the operations involved.

2 Data Transformations

Data transformations in PyTorch preprocess images to make them suitable for training machine learning models. This section explains the transformations used, their purposes, and their effects.

2.1 What Are Transformations?

Transformations are operations applied to images to:

- Standardize, augment, or preprocess data for models.
- Convert images to tensors (a format models understand).
- Normalize data for faster and more stable training.
- Augment data (e.g., flipping, rotating) to improve model generalization.

2.2 transforms.Compose

The `transforms.Compose` utility chains multiple transformations, applying them sequentially. It acts like a pipeline: input data → transformation 1 → transformation 2 → output.

2.3 transforms.ToTensor

This transformation:

- Converts an image (PIL or NumPy format) to a PyTorch tensor.
- Scales pixel values from [0, 255] to [0, 1].
- Rearranges dimensions from (height, width, channels) to (channels, height, width).
- Example: A 224x224 RGB image becomes a tensor of shape (3, 224, 224).

```
import torchvision.transforms as transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Listing 1: Data Transformation Code

2.4 transforms.Normalize

This transformation normalizes a tensor by subtracting a mean and dividing by a standard deviation per channel (R, G, B for images).

- Formula: $output = \frac{input - mean}{std}$
- Parameters (from code):
  - Mean: (0.5, 0.5, 0.5) for R, G, B.
  - Std: (0.5, 0.5, 0.5) for R, G, B.
- Effect: Maps values from [0, 1] to [–1, 1].

2.5 Output of Transformations

When an image passes through the transformations in Listing 1:

1. It becomes a tensor with values in [0, 1] (via `ToTensor`).
2. It is normalized to [–1, 1] per channel (via `Normalize`).
3. The result is a preprocessed tensor ready for a neural network.

2.6 Why Normalize?

Normalization:

- Stabilizes and accelerates training.
- Helps neural networks handle data with large or inconsistent ranges.
- The (0.5, 0.5, 0.5) mean/std is a general choice, unlike ImageNet's (0.485, 0.456, 0.406) mean and (0.229, 0.224, 0.225) std.

2.7 Video Resources

The following videos provide further insights into PyTorch transformations:

- "Image Preprocessing in PyTorch — torchvision.transforms" by Aladdin Persson
- "PyTorch Transformations to Augment Image Training Data" by Jeff Heaton
- "Image Augmentation for Deep Learning" by A Data Odyssey
- "Processing Image Data for Deep Learning" by Siddhardhan

2.8 Summary

A PyTorch transformation preprocesses data for model training by:

- Converting to tensors (`ToTensor`).
- Normalizing to a range like [–1, 1] (`Normalize`).
- Supporting augmentations (e.g., resizing, cropping).

`transforms.Compose` chains these operations for model-ready data.

3 CIFAR-10 Dataset Operations

The CIFAR-10 dataset contains 60,000 32x32 RGB images across 10 classes (e.g., plane, car, bird). This section describes how to load and prepare the dataset.

3.1 Loading the Training Set

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
    transform=transform)
```

Listing 2: Loading CIFAR-10 Training Set

Purpose: Loads 50,000 training images. **Breakdown**:

- `torchvision.datasets.CIFAR10`: PyTorch's tool for CIFAR-10.
- `root='./data'`: Stores data in a "data" folder.
- `train=True`: Selects training images.
- `download=True`: Downloads the dataset if missing.
- `transform=transform`: Applies transformations from Listing 1.

3.2 Creating the Training DataLoader

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True,
    num_workers=0)
```

Listing 3: Training DataLoader

Purpose: Feeds training data to the model in batches. **Breakdown**:

- `DataLoader`: Handles data in chunks.
- `trainset`: Uses the training data from Listing 2.
- `batch_size=32`: Processes 32 images at a time.
- `shuffle=True`: Randomizes image order for better training.
- `num_workers=0`: Loads data using the main process.

3.3 Loading the Test Set

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

Listing 4: Loading CIFAR-10 Test Set

Purpose: Loads 10,000 test images. **Breakdown**: Same as Listing 2, except:

- `train=False`: Selects test images for evaluation.

3.4 Creating the Test DataLoader

```
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False,
    num_workers=0)
```

Listing 5: Test DataLoader

Purpose: Feeds test data to the model in batches. **Breakdown**: Same as Listing 3, except:

- `testset`: Uses test data from Listing 4.
- `shuffle=False`: Keeps test images in order.

4 CNN Model Definition

This section describes the CNN model used to classify CIFAR-10 images into 10 classes.

4.1 Class Definition

```
class CNN(nn.Module):
```

Listing 6: CNN Class Definition

Purpose: Defines a `CNN` class inheriting from `nn.Module`. **Explanation**:

- `class CNN`: Names the model.
- `nn.Module`: Provides tools for layers, parameters, and training.
- Acts as a blueprint for the neural network.

4.2 Initializer

```
def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
    self.fc1 = nn.Linear(64 * 8 * 8, 128)
    self.fc2 = nn.Linear(128, 10)
```

Listing 7: CNN Initializer

Purpose: Initializes the model's layers. **Explanation**:

- `super(CNN, self).__init__()`: Ensures `nn.Module` sets up properly.
- `conv1`: Convolutional layer for RGB images (3 channels), outputs 32 feature maps using 3x3 filters, `padding=1` keeps size at 32x32.
- `pool`: Max pooling layer shrinks images (e.g., 32x32 → 16x16) using 2x2 patches.
- `conv2`: Takes 32 feature maps, outputs 64 with 3x3 filters, `padding=1` maintains size.
- `fc1`: Fully connected layer processes flattened data (64 * 8 * 8 = 4,096 inputs), outputs 128 values.
- `fc2`: Maps 128 values to 10 class scores.

4.3 Forward Pass

```
def forward(self, x):
    x = self.pool(torch.relu(self.conv1(x)))  # 32x32 -> 16x16
    x = self.pool(torch.relu(self.conv2(x)))  # 16x16 -> 8x8
    x = x.view(-1, 64 * 8 * 8)                # Flatten
    x = torch.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

Listing 8: CNN Forward Pass

Purpose: Defines how data flows through the model. **Explanation**:

- `conv1 + relu + pool`: Processes image to 32x16x16.
- `conv2 + relu + pool`: Processes to 64x8x8.
- `view`: Flattens to (batch_size, 4,096).
- `fc1 + relu`: Reduces to 128 values.
- `fc2`: Outputs 10 class scores.
- `return x`: Returns scores for prediction.

5 Model Training and Saving

This section explains how to instantiate, train, and save the CNN model.

5.1 Instantiation and Setup

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Listing 9: Model Setup

Purpose: Sets up the model, loss function, and optimizer. **Explanation**:

- `device`: Selects GPU (`cuda`) if available, else CPU.
- `model`: Instantiates `CNN` and moves it to `device`.
- `criterion`: Uses `CrossEntropyLoss` for classification.
- `optimizer`: Uses Adam optimizer with learning rate 0.001 to update `model.parameters()`.

5.2 Training Loop

```
for epoch in range(5):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 100 == 99:  # print every 100 mini-batches
            print(f"[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}")
            running_loss = 0.0
print("Finished Training")
```

Listing 10: Training Loop

Purpose: Trains the model for 5 epochs. **Explanation**:

- `for epoch in range(5)`: Loops through dataset 5 times.
- `running_loss = 0.0`: Tracks loss per epoch.
- `enumerate(trainloader, 0)`: Iterates over batches (32 images each).
- `inputs, labels`: Moves batch data to `device`.
- `zero_grad()`: Clears old gradients.
- `model(inputs)`: Gets predictions.
- `criterion`: Computes loss.
- `loss.backward()`: Calculates gradients.
- `optimizer.step()`: Updates weights.
- `running_loss`: Accumulates loss.
- `if i % 100 == 99`: Prints average loss every 100 batches.
- `print("Finished Training")`: Signals completion.

5.3 Saving the Model

```
project_root = os.path.dirname(os.path.dirname(__file__))
models_dir = os.path.join(project_root, 'models')
os.makedirs(models_dir, exist_ok=True)
model_path = os.path.join(models_dir, 'cnn_cifar10.pth')
torch.save(model.state_dict(), model_path)
```

```
print(f"    Model saved at {model_path}")
```

Listing 11: Saving Model

Purpose: Saves the trained model. **Explanation**:

- `project_root`: Finds project root directory.
- `models_dir`: Sets path for "models" folder.
- `os.makedirs`: Creates folder if missing.
- `model_path`: Defines file path (`cnn_cifar10.pth`).
- `torch.save`: Saves model weights.
- `print`: Confirms save location.

6 Conclusion

This document detailed the process of building and training a CNN for CIFAR-10 classification using PyTorch, covering data transformations, dataset handling, model architecture, and training. The provided code and explanations serve as a foundation for further exploration in deep learning.