Building and Training a Simple CNN for Image Classification with PyTorch

Kaiser Hamid

December 16, 2024

Contents

1 Introduction

In this article, we will demonstrate how to build and train a simple Convolutional Neural Network (CNN) for image classification using **PyTorch**. We will use a directory structure similar to the previous TensorFlow/Keras example:

```
dataset/
    train/
        dog/
            dog001.jpg
            dog002.jpg
            ...
        cat/
            cat001.jpg
            cat002.jpg
            ...
    val/
        dog/
            dog101.jpg
            dog102.jpg
            ...
        cat/
            cat101.jpg
            cat102.jpg
            ...
```

We have two classes: `dog` and `cat`. The `val/` directory determines the validation set size by the number of images it contains.

We will break down the code into chunks with explanations, similar to the previous article, but now using PyTorch's `torchvision` and `torch.utils.data` utilities.

2 Setup and Data Loading

Chunk 1: Imports and Paths

```
import os
import torch
from torch import nn, optim
from torchvision import datasets, transforms
```

```python
from torch.utils.data import DataLoader
```

Explanation: We import necessary PyTorch libraries:

- `torch`, `nn`, `optim` for building and training neural networks.
- `torchvision.datasets` and `torchvision.transforms` to load and transform images.
- `DataLoader` to create iterable datasets.

Chunk 2: Directory Paths and Parameters

```python
train_dir = 'dataset/train'
val_dir = 'dataset/val'

img_height = 150
img_width = 150
batch_size = 32
```

Explanation: We specify the paths to the training and validation directories and define the image size and batch size. Images will be resized to 150×150 and processed in batches of 32.

Chunk 3: Data Transforms

```python
# Data augmentation and normalization for training
train_transforms = transforms.Compose([
    transforms.Resize((img_height, img_width)),
    transforms.RandomRotation(20),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5],
                         [0.5, 0.5, 0.5])
])

# Just resizing and normalizing for validation
val_transforms = transforms.Compose([
    transforms.Resize((img_height, img_width)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5],
                         [0.5, 0.5, 0.5])
])
```

Explanation: For the training set, we apply transformations including random rotation and horizontal flips to augment the data. For both training and validation sets, we convert images to tensors and normalize pixel values.

Chunk 4: Create Datasets and DataLoaders

```
train_dataset = datasets.ImageFolder(train_dir,
   transform=train_transforms)
val_dataset = datasets.ImageFolder(val_dir, transform=
   val_transforms)

train_loader = DataLoader(train_dataset, batch_size=
   batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=
   batch_size, shuffle=False)
```

Explanation:

- `ImageFolder` automatically assigns labels based on directory names.
- `DataLoader` creates iterable objects. Setting `shuffle=True` for the training loader helps the model generalize better.
- The number of images in `val/` determines the validation set size.

3 Defining the CNN Model

Chunk 5: Defining the CNN Architecture

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, 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.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        # After three times pooling (each halving the
           dimension),
        # starting from 150x150:
        # 1st pool: 75x75
        # 2nd pool: 37x37
        # 3rd pool: 18x18 (since we can't have half a
           pixel, it truncates down)
        # For simplicity, let's assume 18x18 after all
           pools.

        self.fc1 = nn.Linear(128*18*18, 128)
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))

        x = x.view(x.size(0), -1) # Flatten
        x = torch.relu(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x

model = SimpleCNN()
```

Explanation:

- We define a simple CNN with three convolutional layers followed by pooling.
- Each convolution increases the depth (3→32→64→128).
- After flattening, we have two fully connected layers, ending with a single output neuron and a sigmoid activation for binary classification.
- The exact spatial dimension after pooling depends on the input size and kernels. We approximate the final spatial size to 18x18 after three pools for simplicity.

4 Training Setup

Chunk 6: Device, Loss, Optimizer

```
device = torch.device('cuda' if torch.cuda.is_available
   () else 'cpu')
model.to(device)
```

```
criterion = nn.BCELoss()  # Binary Cross Entropy Loss
   for binary classification
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Explanation:

- We use GPU if available, otherwise CPU.
- Use BCELoss since we are dealing with a binary classification problem. Our final layer uses a sigmoid, so BCE is suitable.
- The Adam optimizer adjusts learning rates adaptively.

5 Training the Model

Chunk 7: Training Loop

```
epochs = 10

for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    running_acc = 0.0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(
           device).float().view(-1,1)

        optimizer.zero_grad()
        outputs = model(images)

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

        running_loss += loss.item()
        preds = (outputs > 0.5).float()
        running_acc += (preds == labels).float().mean().
           item()

    train_loss = running_loss/len(train_loader)
    train_acc = running_acc/len(train_loader)
    model.eval()
    val_loss = 0.0
    val_acc = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.
               to(device).float().view(-1,1)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            preds = (outputs > 0.5).float()
            val_acc += (preds == labels).float().mean().
               item()

    val_loss = val_loss/len(val_loader)
    val_acc = val_acc/len(val_loader)

    print(f"Epoch [{epoch+1}/{epochs}]",
          f"Train Loss: {train_loss:.4f}, Train Acc: {
             train_acc:.4f}",
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc
             :.4f}")
```

Explanation:

- For each epoch, we switch to training mode (`model.train()`), iterate over the training data, compute the loss, and update the model parameters.
- After training, we switch to evaluation mode (`model.eval()`) and calculate the validation loss and accuracy.
- We print the results for each epoch.

6 Saving the Model

Chunk 8: Saving the Model

```
torch.save(model.state_dict(), 'simple_cnn_dog_cat.pth')
```

Explanation:

- We save the trained model's parameters to a file for later use or inference.

7 Summary of the Process

- We organized the dataset into a directory structure with separate class folders.
- Used `ImageFolder` to load images and `DataLoader` to create iterable datasets.
- Applied transformations and normalization using `transforms`.
- Defined a simple CNN with PyTorch's `nn.Module`.
- Trained the model on the GPU if available.
- Evaluated performance using a validation set determined by the images placed in the `val` directory.
- Saved the model weights to a file.

This gives a solid starting point for building and training CNNs in PyTorch for image classification.