CNN Architecture Documentation

PyTorch Implementation Guide

September 16, 2025

Contents

1 Introduction

This document provides a comprehensive step-by-step breakdown of a Convolutional Neural Network (CNN) implementation in PyTorch for fruit classification. We'll examine each component from data loading to model training.

2 Imports and Setup

We begin by importing the essential PyTorch libraries and setting up our computational environment.

Listing 1: Library Imports and Device Configuration

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

device = torch.device("cpu")
```

2.1 Library Functions

- `torch` – Core PyTorch library for tensor operations
- `nn` – Neural network building blocks (Conv2d, Linear, etc.)
- `F` – Functional operations (ReLU, softmax, etc.)
- `DataLoader` – Efficient batch loading and shuffling
- `datasets` – Pre-built dataset classes like ImageFolder
- `transforms` – Image preprocessing and augmentation tools

Note: The device is set to CPU. For GPU acceleration, use:
`device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`

3 Data Preprocessing

Image preprocessing is crucial for consistent model input and better convergence.

Listing 2: Image Transformation Pipeline

```
transform = transforms.Compose([
    transforms.Resize((64,64)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
])
```

3.1 Transformation Steps

1. **Resize(64,64):** Standardizes all images to 64×64 pixels
2. **ToTensor():** Converts PIL images to PyTorch tensors (values 0-1)

3. **Normalize():** Transforms values from [0,1] to [-1,1] using:

$$\text{normalized} = \frac{\text{pixel} - 0.5}{0.5} \tag{1}$$

4 Dataset Loading

We use PyTorch's ImageFolder class to automatically organize our fruit dataset.

Listing 3: Dataset Configuration

```
train_dir = '~/datasets/fruits/fruits-360_100x100/fruits-360/Training'
test_dir  = '~/datasets/fruits/fruits-360_100x100/fruits-360/Test'

train_dataset = datasets.ImageFolder(train_dir, transform = transform)
test_dataset = datasets.ImageFolder(test_dir, transform = transform)
```

ImageFolder expects images organized in class-specific subfolders and automatically assigns labels based on folder names.

5 Data Loading

DataLoaders handle batch processing and data shuffling efficiently.

Listing 4: Batch Processing Setup

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

Parameters:

- **batch_size = 64:** Process 64 images per iteration
- **shuffle=True:** Randomize order for better training

6 Class Detection

Listing 5: Automatic Class Counting

```
num_classes = len(train_dataset.classes)
print(f"Number of classes :{num_classes}")
```

This automatically counts the number of fruit categories from the folder structure.

7 CNN Architecture

The SimpleCNN class defines our neural network architecture.

Listing 6: CNN Model Definition

```
class SimpleCNN(nn.Module):
    def __init__(self,num_classes):
        super(SimpleCNN,self).__init__()
        self.conv1 = nn.Conv2d(3,16,kernel_size=3,padding=1)   # input:
            3 channels, output:16
        self.conv2 = nn.Conv2d(16,32,kernel_size=3,padding=1)  # input
            :16 channels, output:32
```

```
        self.pool = nn.MaxPool2d(2,2)                                    # reduces
            H,W by 2
        self.fc1 = nn.Linear(32*16*16,128)                               # flatten
            to 32*16*16
        self.fc2 = nn.Linear(128,num_classes)                            # final
           output layer

    def forward(self,X):
        X = self.pool(F.relu(self.conv1(X)))  # conv1 -> ReLU -> pool
        X = self.pool(F.relu(self.conv2(X)))  # conv2 -> ReLU -> pool
        X = X.view(X.size(0),-1)              # flatten for fully
           connected layer
        X = F.relu(self.fc1(X))               # hidden fully connected
           layer
        X = self.fc2(X)                        # output layer (logits)
        return X
```

7.1 Architecture Flow

The network processes data through these sequential stages:

Input: 64×64×3 RGB Image

↓

Conv1: 3→16 channels + ReLU + MaxPool

↓

Feature Maps: 32×32×16

↓

Conv2: 16→32 channels + ReLU + MaxPool

↓

Feature Maps: 16×16×32

↓

Flatten: 8192 features

↓

FC1: 8192→128 + ReLU

↓

FC2: 128→num_classes (Output)

7.2 Layer Analysis

Layer	Input Shape	Output Shape	Parameters
Input	–	64×64×3	0
Conv1	64×64×3	64×64×16	448
MaxPool1	64×64×16	32×32×16	0
Conv2	32×32×16	32×32×32	4,640
MaxPool2	32×32×32	16×16×32	0
FC1	8,192	128	1,048,704
FC2	128	num_classes	128 × classes

Table 1: Layer-wise Architecture Breakdown

Key Points:

- Convolutional layers extract spatial features

- MaxPooling reduces dimensions while preserving important information
- Fully connected layers perform final classification
- Total flattened size: $32 \times 16 \times 16 = 8{,}192$ features

8 Training Configuration

Listing 7: Model and Training Setup

```
model = SimpleCNN(num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
num_epochs = 5
```

8.1 Configuration Details

- **CrossEntropyLoss:** Standard loss for multi-class classification
- **Adam Optimizer:** Adaptive learning with momentum
- **Learning Rate = 0.001:** Conservative but stable rate
- **Epochs = 5:** Quick training for initial testing

9 Training Loop

Listing 8: Model Training Process

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()       # clear previous gradients
        outputs = model(images)     # forward pass
        loss = criterion(outputs, labels)  # compute loss
        loss.backward()             # backpropagation
        optimizer.step()            # update weights
        running_loss += loss.item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(
        train_loader):.4f}")
```

9.1 Training Steps

The training follows this standard deep learning workflow:

1. **Zero Gradients:** Clear gradients from previous iteration
2. **Forward Pass:** Compute model predictions
3. **Loss Calculation:** Compare predictions with true labels
4. **Backward Pass:** Compute gradients via backpropagation
5. **Parameter Update:** Apply optimizer to update weights
6. **Loss Tracking:** Monitor training progress

10 Bug Fixes and Improvements

10.1 Identified Issues

1. **Typo:** `num_eporchs` should be `num_epochs`
2. **Path Issue:** Tilde (~) may not expand in all environments
3. **Missing Validation:** No validation loop to monitor overfitting

10.2 Recommended Solutions

Fix Path Expansion:

```
import os
train_dir = os.path.expanduser('~/datasets/fruits/fruits-360_100x100/
    fruits-360/Training')
test_dir = os.path.expanduser('~/datasets/fruits/fruits-360_100x100/
    fruits-360/Test')
```

Add Validation Loop:

```
# Add after training loop
model.eval()
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)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

11 Performance Optimization

11.1 Training Improvements

- Implement learning rate scheduling
- Add early stopping mechanism
- Use data augmentation (rotation, flipping, cropping)
- Add batch normalization for stable training
- Implement dropout for regularization

11.2 Architecture Enhancements

- Add more convolutional layers for deeper feature extraction
- Experiment with different kernel sizes
- Try residual connections (ResNet-style)
- Use pre-trained models for transfer learning
- Implement attention mechanisms

12 Conclusion

This CNN implementation provides a solid foundation for image classification tasks. The architecture effectively combines convolutional feature extraction with fully connected classification layers.

Key Takeaways:

- Proper data preprocessing ensures consistent model input
- Convolutional layers learn hierarchical visual features
- Pooling operations provide translation invariance and reduce computation
- The training loop follows standard gradient descent optimization
- Regular validation prevents overfitting

The documented code is ready for implementation and can serve as a baseline for more advanced architectures and techniques.

Created by **Mohamed Amine**

GitHub: https://github.com/Amine-DevAI
LinkedIn: https://www.linkedin.com/in/mohamed-amine-mammar-el-hadj-715a41295

September 16, 2025