Day 3: Deep Learning for Computer Vision

CNNs + Transfer Learning + Training Your First Model

made with ♡, by Aryan

1 Session Overview

Welcome to Day 3! Today we enter the world of deep learning for computer vision. You'll learn why deep learning revolutionized CV, understand CNN architecture, train your first neural network, and leverage transfer learning to build powerful models with limited data.

2 Learning Objectives

- Understand why deep learning outperforms classical methods
- Master CNN architecture components: convolution, pooling, activation
- Calculate output dimensions for CNN layers
- Build and train a CNN from scratch on CIFAR-10
- Apply transfer learning with pretrained models
- Debug common training issues
- Create a Cat vs Dog classifier

3 Part 1: Why Deep Learning?

Deep learning has revolutionized computer vision since 2012 when AlexNet won ImageNet. But why does it work so well?

3.1 Deep Learning vs Classical Methods

Let's compare the two approaches on a real task: **Cat vs Dog classification**

Classical Approach (Day 2):

1. Extract hand-crafted features (HOG, SIFT, color histograms)
2. Features are generic, not task-specific
3. Train a simple classifier (SVM, Random Forest)
4. Accuracy: 70-75%

Deep Learning Approach (Today):

1. Feed raw RGB pixels to CNN
2. Network learns optimal features automatically
3. Features are task-specific
4. Accuracy: 95-98%

3.2 Key Advantages

1. Automatic Feature Learning

Classical methods require hand-crafted features (SIFT, HOG, etc.). Deep learning automatically learns optimal features from data.

2. Hierarchical Representations
CNNs learn features at multiple levels:

- Early layers: edges, colors, basic textures
- Middle layers: parts, patterns (eyes, ears, wheels)
- Deep layers: complete objects, scenes (cat face, car)

Example: A cat classifier learns:

- Layer 1: Edge detectors
- Layer 2: Fur textures, whisker patterns
- Layer 3: Ears, eyes, nose shapes
- Layer 4: Complete cat faces

3. Performance Scaling

- Classical methods plateau with more data
- Deep learning improves with more data and compute
- State-of-the-art results on most vision tasks

3.3 When to Use What?

Criterion	Classical	Deep Learning
Small dataset (< 1000 images)	✓	Maybe
Large dataset ($> 10{,}000$ images)		✓✓
Limited compute/GPU	✓	
Need very fast inference	✓	
State-of-the-art accuracy		✓✓

4 Part 2: CNN Building Blocks

4.1 1. Convolutional Layers

The core of CNNs. Convolution applies filters (kernels) across the image to detect features.

```
import torch
import torch.nn as nn

# Example: Single convolutional layer
conv = nn.Conv2d(
    in_channels=3,    # RGB input
    out_channels=32,  # 32 filters
    kernel_size=3,    # 3x3 filters
    stride=1,         # Move 1 pixel at a time
    padding=1         # Pad to keep size same
)

# Input: batch of RGB images
x = torch.randn(16, 3, 32, 32)  # (batch, channels, height, width)
output = conv(x)
print(output.shape)  # (16, 32, 32, 32) - 32 feature maps
```

Teaching Tip: Draw on board how a 3×3 filter slides across an image, computing dot products at each position.

4.2 2. Understanding Padding

Problem: Convolution shrinks feature maps!

```
Input:  32×32 image
Filter: 3×3, stride=1, no padding
Output: 30×30  <- Lost 2 pixels on each side!

After 5 layers: 32 → 30 → 28 → 26 → 24 → 22
```

Solution: Add padding (extra pixels around border)

Types:

- **Valid** (no padding): Output shrinks
- **Same** (padding=1 for 3×3): Output size = Input size

4.3 3. Output Size Calculation

Formula:

$$\text{Output} = \left\lfloor \frac{\text{Input} - \text{Kernel} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1 \tag{1}$$

Examples:

Input	Kernel	Padding	Stride	Output
32	3	0	1	30
32	3	1	1	32
32	5	2	1	32
32	3	1	2	16
224	7	3	2	112

4.4 4. Activation Functions

Add non-linearity to enable learning complex patterns.

```python
# ReLU: max(0, x) - most common
relu = nn.ReLU()
x = torch.tensor([-2, -1, 0, 1, 2])
print(relu(x))  # tensor([0, 0, 0, 1, 2])
```

Why ReLU?

- Simple and fast
- Helps gradients flow during training
- Prevents vanishing gradient problem

4.5 5. Pooling Layers

Downsample feature maps to reduce computation and add robustness.

```python
# Max pooling: take maximum value in each region
pool = nn.MaxPool2d(kernel_size=2, stride=2)
x = torch.randn(1, 32, 32, 32)
output = pool(x)
print(output.shape)  # (1, 32, 16, 16) - halved dimensions
```

Purpose:

- Reduce spatial dimensions (faster computation)
- Make features more robust to small shifts
- Increase receptive field

4.6 6. Complete CNN Architecture

```python
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)

        # Pooling layer
        self.pool = nn.MaxPool2d(2, 2)

        # Fully connected layers
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, num_classes)

        # Dropout for regularization
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # Block 1: Conv -> ReLU -> Pool
        x = self.pool(F.relu(self.conv1(x)))  # 32x32 -> 16x16

        # Block 2
        x = self.pool(F.relu(self.conv2(x)))  # 16x16 -> 8x8

        # Block 3
        x = self.pool(F.relu(self.conv3(x)))  # 8x8 -> 4x4

        # Flatten: Convert 3D features to 1D
        x = x.view(-1, 128 * 4 * 4)

        # Fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

5 Part 3: Training a CNN on CIFAR-10

```python
import torch
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. Prepare data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

```

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

# 2. Initialize model, loss, optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleCNN(num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 3. Training loop
num_epochs = 10

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)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(
        train_loader):.4f}')
```

5.1 Training Diagnostics: Reading Your Results

Healthy Training:

```
Epoch 1: Loss: 1.82
Epoch 2: Loss: 1.45
Epoch 3: Loss: 1.21  <- Steadily decreasing
...
```

Common Problems:

1. **Loss not decreasing:**
   - Check learning rate (try 0.001, 0.0001)
   - Verify data normalization
   - Check if labels are correct
2. **Loss becomes NaN:**

- Learning rate too high - reduce by 10×
- Check for division by zero

3. **Training accuracy high, test accuracy low:**
   - Overfitting! Add dropout, data augmentation
   - Reduce model size

6 Part 4: Famous CNN Architectures

6.1 ResNet: Going Deeper

Problem: Deeper networks should be better, but they're not!

- 20-layer network: 88% accuracy
- 56-layer network: 82% accuracy ← Worse!

Reason: Vanishing gradients - gradients become tiny in deep networks.
ResNet's Solution: Skip Connections

```
   input (x)
      |
[conv-relu-conv]  <- learns F(x)
      |
      +  <--------- skip connection (adds x)
      |
   output = F(x) + x
```

Why it works:

- Gradients can flow directly through skip connections
- If a layer isn't helpful, it can learn to output 0
- Enables training 100+ layer networks

6.2 Architecture Comparison

Model	Parameters	Accuracy	Use Case
ResNet18	11M	69.8%	Quick baseline
ResNet50	25M	76.1%	General purpose
MobileNetV2	3.5M	72.0%	Mobile/edge devices
EfficientNet-B0	5.3M	77.1%	Best efficiency

How to choose:

- **Learning/prototyping:** ResNet18
- **Best accuracy:** ResNet50 or EfficientNet
- **Mobile deployment:** MobileNetV2

7 Part 5: Transfer Learning

Transfer learning uses models pretrained on ImageNet (1.4M images) as starting points. This dramatically reduces training time and data requirements!

7.1 Why Transfer Learning Works

- Early layers learn generic features (edges, textures) useful for all images
- We only retrain final layers for our specific task
- Requires much less data (100s of images instead of 1000s)
- Trains in minutes instead of hours/days

7.2 Implementation

```
from torchvision import models

# 1. Load pretrained ResNet18
model = models.resnet18(pretrained=True)

# 2. Freeze all layers (don't update them)
for param in model.parameters():
    param.requires_grad = False

# 3. Replace final layer for our task (Cat vs Dog = 2 classes)
model.fc = nn.Linear(model.fc.in_features, 2)

# 4. Only train the final layer
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
```

7.3 Data Transformations for Transfer Learning

Important: Use ImageNet statistics for normalization!

```
# For training: Add augmentation
train_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],  # ImageNet mean
                         [0.229, 0.224, 0.225])  # ImageNet std
])

# For validation/test: No augmentation
val_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

8 Part 6: Learning Rate Scheduling

Learning rate controls how much we update weights. It should start high and decrease over time.

8.1 Why Scheduling Matters

- Start: Large LR to escape bad initialization

- End: Small LR for fine-tuning

8.2 Common Schedules

1. StepLR: Reduce every N epochs

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma
    =0.1)

for epoch in range(num_epochs):
    train_one_epoch()
    scheduler.step()  # LR *= 0.1 every 10 epochs
```

2. ReduceLROnPlateau: Reduce when validation stops improving

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, patience=3, factor=0.5
)

for epoch in range(num_epochs):
    val_loss = validate()
    scheduler.step(val_loss)  # Reduces if val_loss doesn't improve
```

9 Summary

9.1 What You Learned

- ✓Why deep learning beats classical methods
- ✓CNN building blocks: Conv, ReLU, Pooling
- ✓How to calculate output dimensions
- ✓Building and training CNNs from scratch
- ✓Famous architectures: ResNet's skip connections
- ✓Transfer learning for fast, accurate models
- ✓Learning rate scheduling
- ✓Debugging common training issues

9.2 Key Takeaways

1. CNNs automatically learn hierarchical features
2. Transfer learning is your best friend for limited data
3. Always monitor training curves for debugging
4. Start with pretrained models (ResNet18) before building from scratch

10 Assignment Preview

Build a Cat vs Dog classifier using:

- Transfer learning with ResNet18
- Data augmentation (5+ techniques)
- Learning rate scheduling
- Achieve $> 90\%$ accuracy

You're ready to build your first real-world CV model!

Tomorrow: Deploying your model to production!