Modern Architectures: ResNet and EfficientNet

Lecture Overview

This lecture covers modern CNN architectures that enable training very deep networks through residual connections and efficient scaling.

Topics Covered:

- The degradation problem
- ResNet skip connections
- BasicBlock and Bottleneck designs
- EfficientNet compound scaling
- MBConv and squeeze-excitation

1. The Degradation Problem

1.1 Why Deeper Is Not Always Better

```python
import torch
import torch.nn as nn
import numpy as np

print("The Degradation Problem")
print("="*70)

print("Observation:")
print("  56-layer network has HIGHER training error than 20-layer!")
print("  Not overfitting - training error is worse")

print("\nKey insight (He et al., 2015):")
print("  If deeper layers were identity mappings,")
print("  deeper network should be &gt;= as good as shallow")
print("  But plain networks struggle to learn identity!")

print("\nSolution: Residual connections (skip connections)")
print("  Plain:    y = F(x)")
print("  Residual: y = F(x) + x")

print("\nThe network learns F(x) = H(x) - x (the residual)")
print("If identity is optimal, F(x) learns to be zero")
print("Learning zero is easier than learning identity!")
```

```
The Degradation Problem
======================================================================
Observation:
  56-layer network has HIGHER training error than 20-layer!
  Not overfitting - training error is worse

Key insight (He et al., 2015):
  If deeper layers were identity mappings,
  deeper network should be &gt;= as good as shallow
  But plain networks struggle to learn identity!

Solution: Residual connections (skip connections)
  Plain:    y = F(x)
  Residual: y = F(x) + x

The network learns F(x) = H(x) - x (the residual)
If identity is optimal, F(x) learns to be zero
Learning zero is easier than learning identity!
```

2. ResNet Architecture

2.1 Basic Block

```
print("ResNet Basic Block")
print("="*70)

class BasicBlock(nn.Module):
    """For ResNet-18, 34"""
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_ch)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, 1, stride=stride, bias=False),
                nn.BatchNorm2d(out_ch)
            )

    def forward(self, x):
        identity = x
        out = torch.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(identity)  # Skip connection!
        return torch.relu(out)

block = BasicBlock(64, 64)
x = torch.randn(1, 64, 32, 32)
print(f"Input: {x.shape}")
print(f"Output: {block(x).shape}")

print("\nGradient flow through residual:")
print("  dy/dx = dF/dx + 1")
print("  The '+1' ensures gradient always flows!")
```

```
ResNet Basic Block
======================================================================
Input: torch.Size([1, 64, 32, 32])
Output: torch.Size([1, 64, 32, 32])

Gradient flow through residual:
  dy/dx = dF/dx + 1
  The '+1' ensures gradient always flows!
```

2.2 Bottleneck Block

```
print("Bottleneck Block (ResNet-50, 101, 152)")
print("="*70)

print("Design: 1x1 reduce -&gt; 3x3 process -&gt; 1x1 expand")

class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_ch, mid_ch, stride=1):
        super().__init__()
        out_ch = mid_ch * self.expansion
        self.conv1 = nn.Conv2d(in_ch, mid_ch, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(mid_ch)
        self.conv2 = nn.Conv2d(mid_ch, mid_ch, 3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(mid_ch)
        self.conv3 = nn.Conv2d(mid_ch, out_ch, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_ch)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, 1, stride=stride, bias=False),
                nn.BatchNorm2d(out_ch)
```

```
            )

    def forward(self, x):
        identity = x
        out = torch.relu(self.bn1(self.conv1(x)))
        out = torch.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(identity)
        return torch.relu(out)

# Parameter comparison
basic = 2 * (3*3*256*256)
bottleneck = (1*1*256*64) + (3*3*64*64) + (1*1*64*256)
print(f"\nParams (256 channels):")
print(f"  BasicBlock: {basic:,}")
print(f"  Bottleneck: {bottleneck:,}")
print(f"  Reduction: {basic/bottleneck:.1f}x")
```

```
Bottleneck Block (ResNet-50, 101, 152)
======================================================================
Design: 1x1 reduce -&gt; 3x3 process -&gt; 1x1 expand

Params (256 channels):
  BasicBlock: 1,179,648
  Bottleneck: 69,632
  Reduction: 16.9x
```

2.3 ResNet Variants

```
print("ResNet Variants")
print("="*70)

print(f"{'Model':&lt;12} {'Blocks':&gt;20} {'Params':&gt;10}")
print("-" * 45)
print(f"{'ResNet-18':&lt;12} {'[2, 2, 2, 2]':&gt;20} {'11.7M':&gt;10}")
print(f"{'ResNet-34':&lt;12} {'[3, 4, 6, 3]':&gt;20} {'21.8M':&gt;10}")
print(f"{'ResNet-50':&lt;12} {'[3, 4, 6, 3]*':&gt;20} {'25.6M':&gt;10}")
print(f"{'ResNet-101':&lt;12} {'[3, 4, 23, 3]*':&gt;20} {'44.5M':&gt;10}")
print(f"{'ResNet-152':&lt;12} {'[3, 8, 36, 3]*':&gt;20} {'60.2M':&gt;10}")
print("* uses Bottleneck blocks")

print("\nUsing pre-trained ResNet:")
print("  from torchvision import models")
print("  model = models.resnet50(pretrained=True)")
print("  model.fc = nn.Linear(2048, num_classes)")
```

```
ResNet Variants
======================================================================
Model              Blocks     Params
---------------------------------------------
ResNet-18        [2, 2, 2, 2]      11.7M
ResNet-34        [3, 4, 6, 3]      21.8M
ResNet-50       [3, 4, 6, 3]*      25.6M
ResNet-101     [3, 4, 23, 3]*      44.5M
ResNet-152      [3, 8, 36, 3]*      60.2M
* uses Bottleneck blocks

Using pre-trained ResNet:
  from torchvision import models
  model = models.resnet50(pretrained=True)
  model.fc = nn.Linear(2048, num_classes)
```

3. EfficientNet

3.1 Compound Scaling

```
print("EfficientNet (Tan &amp; Le, 2019)")
print("="*70)

print("Key insight: Scale depth, width, resolution together")

print("\nScaling dimensions:")
print("  Depth: Number of layers")
print("  Width: Number of channels")
print("  Resolution: Input image size")

print("\nCompound scaling:")
print("  depth = alpha^phi")
print("  width = beta^phi")
print("  resolution = gamma^phi")
print("  Constraint: alpha * beta^2 * gamma^2 = 2")

print("\nEfficientNet Family:")
print(f"{'Model':&lt;8} {'Params':&gt;8} {'Top-1':&gt;8}")
print("-" * 26)
for m, p, a in [("B0","5.3M","77.1%"),("B1","7.8M","79.1%"),
                ("B2","9.2M","80.1%"),("B3","12M","81.6%"),
                ("B4","19M","82.9%"),("B7","66M","84.3%")]:
    print(f"{m:&lt;8} {p:&gt;8} {a:&gt;8}")

print("\nComparison:")
print("  ResNet-50: 25.6M params, 76% Top-1")
print("  EfficientNet-B0: 5.3M params, 77% Top-1")
print("  5x fewer params, better accuracy!")
```

```
EfficientNet (Tan &amp; Le, 2019)
======================================================================
Key insight: Scale depth, width, resolution together

Scaling dimensions:
  Depth: Number of layers
  Width: Number of channels
  Resolution: Input image size

Compound scaling:
  depth = alpha^phi
  width = beta^phi
  resolution = gamma^phi
  Constraint: alpha * beta^2 * gamma^2 = 2

EfficientNet Family:
Model      Params    Top-1
--------------------------
B0           5.3M    77.1%
B1           7.8M    79.1%
B2           9.2M    80.1%
B3            12M    81.6%
B4            19M    82.9%
B7            66M    84.3%

Comparison:
  ResNet-50: 25.6M params, 76% Top-1
  EfficientNet-B0: 5.3M params, 77% Top-1
  5x fewer params, better accuracy!
```

3.2 MBConv Block

```
print("MBConv (Mobile Inverted Bottleneck)")
print("="*70)

print("MBConv structure:")
print("  1. Expand: 1x1 conv (increase channels)")
print("  2. Depthwise: 3x3 or 5x5 depthwise conv")
print("  3. Squeeze-Excitation: Channel attention")
print("  4. Project: 1x1 conv (reduce channels)")
```

```
class SqueezeExcitation(nn.Module):
    def __init__(self, channels, reduction=4):
        super().__init__()
        reduced = max(1, channels // reduction)
        self.fc1 = nn.Linear(channels, reduced)
        self.fc2 = nn.Linear(reduced, channels)

    def forward(self, x):
        b, c, _, _ = x.shape
        y = x.mean(dim=[2, 3])  # Global avg pool
        y = torch.relu(self.fc1(y))
        y = torch.sigmoid(self.fc2(y))
        return x * y.view(b, c, 1, 1)

print("\nSqueeze-Excitation:")
print("  - Global average pool to get channel statistics")
print("  - FC layers to learn channel importance")
print("  - Sigmoid to get attention weights")
print("  - Multiply to reweight channels")

print("\nActivation: SiLU (Swish)")
print("  SiLU(x) = x * sigmoid(x)")
```

```
MBConv (Mobile Inverted Bottleneck)
======================================================================
MBConv structure:
  1. Expand: 1x1 conv (increase channels)
  2. Depthwise: 3x3 or 5x5 depthwise conv
  3. Squeeze-Excitation: Channel attention
  4. Project: 1x1 conv (reduce channels)

Squeeze-Excitation:
  - Global average pool to get channel statistics
  - FC layers to learn channel importance
  - Sigmoid to get attention weights
  - Multiply to reweight channels

Activation: SiLU (Swish)
  SiLU(x) = x * sigmoid(x)
```

Summary

Key Takeaways:

- Skip connections solve the degradation problem
- Residual blocks learn F(x) = H(x) - x
- Bottleneck design reduces parameters
- EfficientNet scales depth, width, resolution together
- MBConv uses depthwise separable convolutions
- Squeeze-Excitation provides channel attention

Practice Exercises:

1. Implement BasicBlock and Bottleneck
2. Train ResNet-18 on CIFAR-10
3. Compare ResNet vs VGG training
4. Visualize gradient flow with skip connections
5. Load and analyze EfficientNet