CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 10, Lecture 20

Pooling, Feature Hierarchies, and Classic Architectures

Lecture Overview

This lecture covers pooling operations, CNN feature hierarchies, and classic architectures that shaped deep learning.

Topics Covered:

- Max and average pooling
- Global average pooling
- Feature hierarchy in CNNs
- LeNet-5, AlexNet, VGGNet

1. Pooling Layers

1.1 Max and Average Pooling

```
import torch
import torch.nn as nn

print("Pooling Layers")
print("="*70)

print("Pooling: Reduce spatial dimensions")
print("  - Provides translation invariance")
print("  - Reduces computation")
print("  - Increases receptive field")

x = torch.tensor([[[[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12],
                    [13, 14, 15, 16]]]], dtype=torch.float32)

print(f"\nInput (4x4):\n{x.squeeze()}")

max_pool = nn.MaxPool2d(kernel_size=2, stride=2)
avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)

print(f"\nMaxPool2d(2x2):\n{max_pool(x).squeeze()}")
print("  Takes maximum in each region")

print(f"\nAvgPool2d(2x2):\n{avg_pool(x).squeeze()}")
print("  Takes average in each region")

print("\nMax vs Average:")
print("  MaxPool: Preserves strongest activations")
print("  AvgPool: Smoother, sometimes at end of network")
```

```
Pooling Layers
======================================================================
Pooling: Reduce spatial dimensions
  - Provides translation invariance
  - Reduces computation
  - Increases receptive field

Input (4x4):
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.],
        [13., 14., 15., 16.]])

MaxPool2d(2x2):
tensor([[ 6.,  8.],
        [14., 16.]])
  Takes maximum in each region

AvgPool2d(2x2):
tensor([[ 3.5,  5.5],
        [11.5, 13.5]])
  Takes average in each region

Max vs Average:
  MaxPool: Preserves strongest activations
  AvgPool: Smoother, sometimes at end of network
```

1.2 Global Average Pooling

```
print("Global Average Pooling (GAP)")
print("="*70)

print("GAP: Reduces each feature map to single value")
print("  Alternative to flatten + FC layers")
print("  Much fewer parameters!")

x = torch.randn(4, 512, 7, 7)
print(f"\nInput: {x.shape}")
```

```
gap = nn.AdaptiveAvgPool2d(1)
out = gap(x).view(x.size(0), -1)
print(f"After GAP: {out.shape}")

print("\nParameter Comparison:")
fc_params = 512 * 7 * 7 * 1000
gap_params = 512 * 1000
print(f"  Flatten + FC: {fc_params:,}")
print(f"  GAP + FC: {gap_params:,}")
print(f"  Reduction: {fc_params/gap_params:.0f}x")
```

```
Global Average Pooling (GAP)
======================================================================
GAP: Reduces each feature map to single value
  Alternative to flatten + FC layers
  Much fewer parameters!

Input: torch.Size([4, 512, 7, 7])
After GAP: torch.Size([4, 512])

Parameter Comparison:
  Flatten + FC: 25,088,000
  GAP + FC: 512,000
  Reduction: 49x
```

2. Feature Hierarchies

2.1 What CNNs Learn

```
print("Feature Hierarchies in CNNs")
print("="*70)

print("CNNs learn hierarchical representations:")

print("\nLayer 1 (Early):")
print("  - Edges at various orientations")
print("  - Color blobs")
print("  - Simple textures")

print("\nLayers 2-3 (Middle):")
print("  - Corners and contours")
print("  - Texture patterns")
print("  - Simple shapes")

print("\nLayers 4-5 (Late):")
print("  - Object parts (eyes, wheels)")
print("  - Complex patterns")

print("\nFinal Layers:")
print("  - Whole objects")
print("  - Scene categories")

class VGGLike(nn.Module):
    def __init__(self):
        super().__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2))
        self.block2 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2))
        self.block3 = nn.Sequential(
            nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2))

    def forward(self, x):
        f1 = self.block1(x)
        f2 = self.block2(f1)
        f3 = self.block3(f2)
        return f1, f2, f3

model = VGGLike()
x = torch.randn(1, 3, 64, 64)
f1, f2, f3 = model(x)

print(f"\nFeature map shapes:")
print(f"  Input: {x.shape}")
print(f"  Block 1: {f1.shape} (edges)")
print(f"  Block 2: {f2.shape} (textures)")
print(f"  Block 3: {f3.shape} (parts)")
```

```
Feature Hierarchies in CNNs
======================================================================
CNNs learn hierarchical representations:

Layer 1 (Early):
  - Edges at various orientations
  - Color blobs
  - Simple textures

Layers 2-3 (Middle):
  - Corners and contours
  - Texture patterns
  - Simple shapes

Layers 4-5 (Late):
  - Object parts (eyes, wheels)
  - Complex patterns

Final Layers:
  - Whole objects
  - Scene categories
```

```
Feature map shapes:
  Input: torch.Size([1, 3, 64, 64])
  Block 1: torch.Size([1, 64, 32, 32]) (edges)
  Block 2: torch.Size([1, 128, 16, 16]) (textures)
  Block 3: torch.Size([1, 256, 8, 8]) (parts)
```

3. Classic CNN Architectures

3.1 LeNet-5

```
print("LeNet-5 (LeCun et al., 1998)")
print("="*70)

print("First successful CNN - digit recognition")

class LeNet5(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 6, 5), nn.ReLU(), nn.AvgPool2d(2),
            nn.Conv2d(6, 16, 5), nn.ReLU(), nn.AvgPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16*5*5, 120), nn.ReLU(),
            nn.Linear(120, 84), nn.ReLU(),
            nn.Linear(84, num_classes)
        )
    def forward(self, x):
        return self.classifier(self.features(x))

model = LeNet5()
params = sum(p.numel() for p in model.parameters())
print(f"Parameters: {params:,}")

print("\nKey Contributions:")
print("  - Local receptive fields")
print("  - Weight sharing")
print("  - Subsampling (pooling)")
```

```
LeNet-5 (LeCun et al., 1998)
======================================================================
First successful CNN - digit recognition
Parameters: 61,706

Key Contributions:
  - Local receptive fields
  - Weight sharing
  - Subsampling (pooling)
```

3.2 AlexNet

```
print("AlexNet (Krizhevsky et al., 2012)")
print("="*70)

print("Won ImageNet 2012 - started deep learning revolution")

print("\nArchitecture (simplified):")
print("  Conv(3-&gt;96, 11x11, stride=4) -&gt; ReLU -&gt; MaxPool")
print("  Conv(96-&gt;256, 5x5) -&gt; ReLU -&gt; MaxPool")
print("  Conv(256-&gt;384, 3x3) -&gt; ReLU")
print("  Conv(384-&gt;384, 3x3) -&gt; ReLU")
print("  Conv(384-&gt;256, 3x3) -&gt; ReLU -&gt; MaxPool")
print("  FC(9216-&gt;4096) -&gt; FC(4096-&gt;4096) -&gt; FC(4096-&gt;1000)")

print("\nParameters: ~61M")

print("\nKey Innovations:")
print("  - First to use ReLU activation")
print("  - Dropout for regularization")
print("  - Heavy data augmentation")
print("  - GPU training (2 GPUs)")
print("  - Local Response Normalization")
```

```
AlexNet (Krizhevsky et al., 2012)
======================================================================
Won ImageNet 2012 - started deep learning revolution

Architecture (simplified):
  Conv(3-&gt;96, 11x11, stride=4) -&gt; ReLU -&gt; MaxPool
  Conv(96-&gt;256, 5x5) -&gt; ReLU -&gt; MaxPool
  Conv(256-&gt;384, 3x3) -&gt; ReLU
  Conv(384-&gt;384, 3x3) -&gt; ReLU
  Conv(384-&gt;256, 3x3) -&gt; ReLU -&gt; MaxPool
  FC(9216-&gt;4096) -&gt; FC(4096-&gt;4096) -&gt; FC(4096-&gt;1000)

Parameters: ~61M

Key Innovations:
  - First to use ReLU activation
  - Dropout for regularization
  - Heavy data augmentation
  - GPU training (2 GPUs)
  - Local Response Normalization
```

3.3 VGGNet

```
print("VGGNet (Simonyan &amp; Zisserman, 2014)")
print("="*70)

print("Key insight: Use small 3x3 filters, stack deep")
print("  Two 3x3 = same receptive field as one 5x5")
print("  Three 3x3 = one 7x7")
print("  But fewer params and more nonlinearities!")

one_7x7 = 7 * 7 * 64 * 64
three_3x3 = 3 * (3 * 3 * 64 * 64)
print(f"\nParameter comparison:")
print(f"  One 7x7: {one_7x7:,}")
print(f"  Three 3x3: {three_3x3:,}")

print("\nVGG16 Architecture:")
print("  64, 64, pool")
print("  128, 128, pool")
print("  256, 256, 256, pool")
print("  512, 512, 512, pool")
print("  512, 512, 512, pool")
print("  FC 4096, FC 4096, FC 1000")

print("\nVGG16 Parameters: ~138M")

print("\nArchitecture Comparison:")
print(f"{'Model':&lt;10} {'Year':&gt;6} {'Depth':&gt;6} {'Params':&gt;10}")
print("-" * 35)
print(f"{'LeNet':&lt;10} {'1998':&gt;6} {'5':&gt;6} {'60K':&gt;10}")
print(f"{'AlexNet':&lt;10} {'2012':&gt;6} {'8':&gt;6} {'61M':&gt;10}")
print(f"{'VGG16':&lt;10} {'2014':&gt;6} {'16':&gt;6} {'138M':&gt;10}")
```

```
VGGNet (Simonyan &amp; Zisserman, 2014)
======================================================================
Key insight: Use small 3x3 filters, stack deep
  Two 3x3 = same receptive field as one 5x5
  Three 3x3 = one 7x7
  But fewer params and more nonlinearities!

Parameter comparison:
  One 7x7: 200,704
  Three 3x3: 110,592

VGG16 Architecture:
  64, 64, pool
  128, 128, pool
  256, 256, 256, pool
  512, 512, 512, pool
  512, 512, 512, pool
  FC 4096, FC 4096, FC 1000

VGG16 Parameters: ~138M

Architecture Comparison:
Model        Year  Depth     Params
-----------------------------------
LeNet        1998      5        60K
AlexNet      2012      8        61M
```

```
VGG16        2014     16       138M
```

Summary

Key Takeaways:

- Pooling reduces spatial dimensions and provides invariance
- GAP reduces parameters vs flatten+FC
- CNNs learn hierarchical features: edges -> textures -> parts -> objects
- LeNet proved CNNs work
- AlexNet started deep learning revolution
- VGG showed small 3x3 filters stacked deep are effective

Practice Exercises:

1. Implement max pooling from scratch
2. Implement LeNet-5 and train on MNIST
3. Calculate receptive fields
4. Visualize feature maps at different depths
5. Compare different pooling strategies