CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 10, Lecture 19

Convolution Operation and CNN Architecture

Lecture Overview

This lecture introduces the convolution operation and basic CNN architecture for image processing.

Topics Covered:

- Motivation for convolutions
- The convolution operation
- Padding and stride
- PyTorch Conv2d layer
- Building simple CNN architectures

1. Introduction to Convolution

1.1 Why Convolutions for Images?

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

print("Why Convolutions for Images?")
print("="*70)

print("Problems with Fully Connected Networks:")
print("  - Too many parameters (224x224x3 = 150,528 inputs)")
print("  - No spatial structure exploitation")
print("  - No translation invariance")

print("\nConvolutional Neural Networks solve these:")
print("  1. Local connectivity: Each neuron sees small region")
print("  2. Parameter sharing: Same weights across locations")
print("  3. Translation equivariance: Detect patterns anywhere")

# Parameter comparison
fc_params = 224 * 224 * 3 * 1000
conv_params = 3 * 3 * 3 * 64

print(f"\nParameter Comparison (224x224 RGB image):")
print(f"  FC to 1000 neurons: {fc_params:,} params")
print(f"  Conv 3x3, 3-&gt;64: {conv_params:,} params")
print(f"  Reduction: {fc_params/conv_params:.0f}x fewer!")
```

```
Why Convolutions for Images?
======================================================================
Problems with Fully Connected Networks:
  - Too many parameters (224x224x3 = 150,528 inputs)
  - No spatial structure exploitation
  - No translation invariance

Convolutional Neural Networks solve these:
  1. Local connectivity: Each neuron sees small region
  2. Parameter sharing: Same weights across locations
  3. Translation equivariance: Detect patterns anywhere

Parameter Comparison (224x224 RGB image):
  FC to 1000 neurons: 150,528,000 params
  Conv 3x3, 3-&gt;64: 1,728 params
  Reduction: 87111x fewer!
```

1.2 The Convolution Operation

```
print("The Convolution Operation")
print("="*70)

def conv2d_manual(image, kernel):
    """Manual 2D convolution"""
    h, w = image.shape
    kh, kw = kernel.shape
    out_h, out_w = h - kh + 1, w - kw + 1
    output = np.zeros((out_h, out_w))

    for i in range(out_h):
        for j in range(out_w):
            patch = image[i:i+kh, j:j+kw]
            output[i, j] = np.sum(patch * kernel)
    return output

# Example: Edge detection
image = np.array([
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
], dtype=float)

sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=float)

print("Input Image (6x6):")
print(image)
print("\nSobel X kernel (vertical edges):")
print(sobel_x)

output = conv2d_manual(image, sobel_x)
print(f"\nOutput shape: {output.shape}")
print(output)
```

```
The Convolution Operation
======================================================================
Input Image (6x6):
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]

Sobel X kernel (vertical edges):
[[-1.  0.  1.]
 [-2.  0.  2.]
 [-1.  0.  1.]]

Output shape: (4, 4)
[[ 0.  1.  1.  0.]
 [ 0.  2.  2.  0.]
 [ 0.  2.  2.  0.]
 [ 0.  1.  1.  0.]]
```

2. Padding and Stride

2.1 Padding

```
print("Padding")
print("="*70)

print("Without padding:")
print("  Output size = Input - Kernel + 1")
print("  Problem: Output shrinks, border info lost")

print("\nWith 'same' padding:")
print("  Pad input with zeros to maintain size")
print("  pad = (kernel_size - 1) // 2")

conv_no_pad = nn.Conv2d(1, 1, kernel_size=3, padding=0)
conv_same = nn.Conv2d(1, 1, kernel_size=3, padding=1)

x = torch.randn(1, 1, 5, 5)
print(f"\nInput shape: {x.shape}")
print(f"Output (no pad): {conv_no_pad(x).shape}")
print(f"Output (pad=1): {conv_same(x).shape}")

print("\nOutput size formula:")
print("  out = (input + 2*padding - kernel) // stride + 1")

def calc_output(inp, k, p, s):
    return (inp + 2*p - k) // s + 1

print("\nExamples:")
for inp, k, p, s in [(32,3,0,1), (32,3,1,1), (32,3,1,2)]:
    print(f"  In={inp}, K={k}, P={p}, S={s} -&gt; Out={calc_output(inp,k,p,s)}")
```

```
Padding
======================================================================
Without padding:
  Output size = Input - Kernel + 1
  Problem: Output shrinks, border info lost

With 'same' padding:
  Pad input with zeros to maintain size
  pad = (kernel_size - 1) // 2

Input shape: torch.Size([1, 1, 5, 5])
Output (no pad): torch.Size([1, 1, 3, 3])
Output (pad=1): torch.Size([1, 1, 5, 5])

Output size formula:
  out = (input + 2*padding - kernel) // stride + 1

Examples:
  In=32, K=3, P=0, S=1 -&gt; Out=30
  In=32, K=3, P=1, S=1 -&gt; Out=32
  In=32, K=3, P=1, S=2 -&gt; Out=16
```

2.2 Stride

```
print("Stride")
print("="*70)

print("Stride = step size when sliding kernel")
print("  Stride=1: Move 1 pixel at a time")
print("  Stride=2: Move 2 pixels (downsamples by 2x)")

conv_s1 = nn.Conv2d(1, 1, kernel_size=3, stride=1, padding=1)
conv_s2 = nn.Conv2d(1, 1, kernel_size=3, stride=2, padding=1)

x = torch.randn(1, 1, 8, 8)
print(f"\nInput shape: {x.shape}")
print(f"Stride=1: {conv_s1(x).shape}")
print(f"Stride=2: {conv_s2(x).shape}")

print("\nStride for Downsampling:")
print("  Alternative to pooling layers")
print("  Modern architectures often prefer strided convs")

conv_down = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)
x = torch.randn(1, 32, 64, 64)
print(f"\nDownsample pattern: {x.shape} -&gt; {conv_down(x).shape}")
```

```
Stride
======================================================================
Stride = step size when sliding kernel
  Stride=1: Move 1 pixel at a time
  Stride=2: Move 2 pixels (downsamples by 2x)

Input shape: torch.Size([1, 1, 8, 8])
Stride=1: torch.Size([1, 1, 8, 8])
Stride=2: torch.Size([1, 1, 4, 4])

Stride for Downsampling:
  Alternative to pooling layers
  Modern architectures often prefer strided convs

Downsample pattern: torch.Size([1, 32, 64, 64]) -&gt; torch.Size([1, 64, 32, 32])
```

3. PyTorch Conv2d and CNN Blocks

3.1 Conv2d Layer

```
print("PyTorch Conv2d Layer")
print("="*70)

conv = nn.Conv2d(
    in_channels=3,
    out_channels=64,
    kernel_size=3,
    stride=1,
    padding=1,
    bias=True
)

print("Conv2d Parameters:")
print(f"  in_channels: 3 (RGB)")
print(f"  out_channels: 64 (64 filters)")
print(f"  kernel_size: 3x3")

print(f"\nWeight shape: {conv.weight.shape}")
print(f"  [out_ch, in_ch, kernel_h, kernel_w]")
print(f"Bias shape: {conv.bias.shape}")
print(f"Total params: {conv.weight.numel() + conv.bias.numel()}")

x = torch.randn(8, 3, 32, 32)
out = conv(x)
print(f"\nInput: {x.shape}")
print(f"Output: {out.shape}")
```

```
PyTorch Conv2d Layer
======================================================================
Conv2d Parameters:
  in_channels: 3 (RGB)
  out_channels: 64 (64 filters)
  kernel_size: 3x3

Weight shape: torch.Size([64, 3, 3, 3])
  [out_ch, in_ch, kernel_h, kernel_w]
Bias shape: torch.Size([64])
Total params: 1792

Input: torch.Size([8, 3, 32, 32])
Output: torch.Size([8, 64, 32, 32])
```

3.2 Standard CNN Block

```
print("Standard CNN Block: Conv -&gt; BN -&gt; ReLU")
print("="*70)

class ConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, kernel_size=3, stride=1):
        super().__init__()
        padding = kernel_size // 2
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size,
                              stride=stride, padding=padding, bias=False)
        self.bn = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        return self.relu(self.bn(self.conv(x)))

block = ConvBlock(3, 64)
print(block)

x = torch.randn(4, 3, 32, 32)
print(f"\nInput: {x.shape}")
print(f"Output: {block(x).shape}")

print("\nNote: bias=False when using BatchNorm")
print("  BN has its own learnable shift (beta)")
```

```
Standard CNN Block: Conv -&gt; BN -&gt; ReLU
======================================================================
ConvBlock(
  (conv): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (relu): ReLU(inplace=True)
)

Input: torch.Size([4, 3, 32, 32])
Output: torch.Size([4, 64, 32, 32])

Note: bias=False when using BatchNorm
  BN has its own learnable shift (beta)
```

4. Simple CNN Architecture

4.1 Building a CNN

```
print("Simple CNN for CIFAR-10")
print("="*70)

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            # Block 1: 3-&gt;32, 32x32-&gt;16x16
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),
            # Block 2: 32-&gt;64, 16x16-&gt;8x8
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),
            # Block 3: 64-&gt;128, 8x8-&gt;4x4
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        return self.classifier(x)

model = SimpleCNN(10)
total = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total:,}")

x = torch.randn(4, 3, 32, 32)
out = model(x)
print(f"\nInput: {x.shape}")
print(f"Output: {out.shape}")
```

```
Simple CNN for CIFAR-10
======================================================================
Total parameters: 610,794

Input: torch.Size([4, 3, 32, 32])
Output: torch.Size([4, 10])
```

Summary

Key Takeaways:

- CNNs exploit spatial structure through local connectivity
- Parameter sharing makes CNNs efficient
- Convolution is sliding window dot product
- Padding controls output size, stride controls downsampling
- Standard pattern: Conv -> BatchNorm -> ReLU -> Pool

Practice Exercises:

1. Implement 2D convolution from scratch
2. Calculate output sizes for various configurations
3. Build a CNN for MNIST classification
4. Experiment with different kernel sizes
5. Visualize learned convolutional filters