Lab 3: Architecture Implementation Mastery

From ResNet to Vision Transformers - Hands-On

Prof. David Olivieri

Artificial Vision (VIAR25/26) - UVigo

September 22, 2025

Lab 3 Overview: Architecture Implementation Mastery

Learning Objectives

By the end of this lab, you will be able to:

- **Implement** ResNet from mathematical principles with proper gradient flow analysis
- **Design** and integrate attention mechanisms (SE, CBAM) into existing architectures
- **Build** Vision Transformer components from scratch with position encoding
- **Compare** CNN vs ViT performance across different data regimes and computational constraints
- **Analyze** architectural trade-offs and make informed deployment decisions

Lab Structure (4 hours)

Part 1: ResNet Deep Dive (90 min)

- Mathematical foundation review
- Basic vs Bottleneck block implementation
- Gradient flow visualization
- Depth scaling experiments

Part 2: Attention Integration (90 min)

- SE module implementation
- CBAM spatial + channel attention
- Performance benchmarking
- Attention visualization

Continued...

Part 3: Vision Transformer (60 min)

- Patch embedding implementation
- Multi-head self-attention
- Complete ViT architecture

Part 4: Comparative Analysis (60 min)

- CNN vs ViT benchmarking
- Computational profiling
- Architecture selection framework

Tools & Environment

- PyTorch 2.0+, torchvision
- CIFAR-10/100 datasets
- GPU recommended (Google Colab OK)
- Weights & Biases for experiment tracking

Section 1

Part 1: ResNet Deep Dive

Part 1: ResNet Implementation Deep Dive

1.1 Mathematical Foundation Review (15 min)

- Derive residual learning formulation: $H(x) = F(x) + x$
- Analyze gradient flow: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial H} \cdot (1 + \frac{\partial F}{\partial x})$
- Identity mapping importance and degradation problem

1.2 Block Implementation (45 min)

- `BasicBlock`: 3×3→3×3 convolutions with skip connection
- `BottleneckBlock`: 1×1→3×3→1×1 with 4× channel reduction
- Pre-activation vs post-activation comparison
- Proper padding and stride handling for dimension matching

1.3 Architecture Scaling (30 min)

- Build ResNet-18, 34, 50, 101 from block components
- Measure training time, memory usage, and convergence
- Plot accuracy vs depth relationship
- Analyze when deeper networks help vs hurt

ResNet Implementation: Key Components

Basic Block Structure

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
                 stride=1, downsample=None):
        # TODO: Implement basic block
        # - Two 3x3 convolutions
        # - Batch normalization
        # - ReLU activation
        # - Skip connection handling
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # Remember: out = F(x) + x
        pass
```

Bottleneck Block Structure

```
class BottleneckBlock(nn.Module):
    expansion = 4

    def __init__(self, in_channels, out_channels,
                 stride=1, downsample=None):
        # TODO: Implement bottleneck block
        # - 1x1 conv (reduce channels)
        # - 3x3 conv (spatial processing)
        # - 1x1 conv (expand channels)
        # - Batch norm + ReLU between each
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        pass
```

Key Implementation Points

- Handle dimension mismatch with 1×1 convolution in skip connection
- Proper stride application only on first convolution of each block
- BatchNorm placement: after conv, before ReLU (post-activation ResNet)
- Expansion factor of 4 for bottleneck blocks

Section 2

Part 2: Attention Integration

Part 2: Attention Mechanism Integration

2.1 SE Module Implementation (30 min)

- Global average pooling: $z_c = \frac{1}{H \times W} \sum_{i,j} x_{i,j,c}$
- Excitation network: $s = \sigma(W_2 \delta(W_1 z))$ with reduction ratio
- Channel-wise multiplication: $\tilde{x}_{i,j,c} = s_c \cdot x_{i,j,c}$
- Integration into ResNet blocks

2.2 CBAM Implementation (30 min)

- Channel attention: Apply SE mechanism
- Spatial attention: $M_s = \sigma(\text{conv}_{77}([\text{AvgPool}(F); \text{MaxPool}(F)]))$
- Sequential application: $F'' = M_s(F') \odot F'$ where $F' = M_c(F) \odot F$

2.3 Performance Analysis (30 min)

- Benchmark ResNet vs ResNet+SE vs ResNet+CBAM
- Measure computational overhead and memory usage
- Visualize attention maps using Grad-CAM
- Analyze which channels/regions receive highest attention

Attention Mechanisms: SE and CBAM

SE Module

```python
class SEModule(nn.Module):
    def __init__(self, channels, reduction=16):
        super().__init__()
        # TODO: Implement SE module
        # - Global Average Pooling
        # - FC layer with reduction
        # - ReLU activation
        # - FC layer back to channels
        # - Sigmoid activation
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # 1. Global avg pool (B,C,H,W) -> (B,C,1,1)
        # 2. Squeeze through FC layers
        # 3. Sigmoid to get weights
        # 4. Scale input: x * weights
        pass
```

CBAM Module

```python
class CBAM(nn.Module):
    def __init__(self, channels, reduction=16):
        super().__init__()
        # TODO: Implement CBAM
        # - Channel attention (SE-like)
        # - Spatial attention
        #   * Avg + Max pool along channel
        #   * 7x7 conv + sigmoid
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # 1. Apply channel attention
        # 2. Apply spatial attention
        # 3. Return attended features
        pass
```

Implementation Tips

- Use `AdaptiveAvgPool2d(1)` for global pooling
- Reduction ratio typically 16 for SE modules
- For CBAM spatial attention, concatenate avg and max pooled features
- Visualize attention maps with `torch.nn.functional.interpolate`

Section 3

Part 3: Vision Transformer

Part 3: Vision Transformer from Scratch

3.1 Patch Embedding Layer (20 min)

- Image to patches: $(H, W, C) \to (N, P^2 \cdot C)$ where $N = \frac{H \times W}{P^2}$
- Linear projection: $\mathbf{x}_i \to \mathbf{e}_i = \mathbf{x}_i \mathbf{W}_e + \mathbf{b}_e$
- Class token prepending: $[\mathbf{x}_{\text{cls}}, \mathbf{e}_1, \ldots, \mathbf{e}_N]$
- Position encoding addition (learnable vs sinusoidal)

3.2 Multi-Head Self-Attention (25 min)

- Query, Key, Value projections for each head
- Scaled dot-product attention: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
- Multi-head concatenation and output projection
- Attention weight visualization and interpretation

3.3 Complete ViT Architecture (15 min)

- Transformer encoder block: LayerNorm $\to$ MSA $\to$ Add $\to$ LayerNorm $\to$ MLP $\to$ Add
- Stack multiple encoder layers
- Classification head: Extract [CLS] token $\to$ Linear layer
- Hyperparameter exploration: patch size, embedding dimension, number of heads

Vision Transformer: Core Components

Patch Embedding

```
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=224, patch_size=16,
                 in_channels=3, embed_dim=768):
        super().__init__()
        # TODO: Implement patch embedding
        # - Calculate number of patches
        # - Conv2d with kernel=patch_size, stride=patch_size
        # - Or use nn.Unfold + Linear
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # (B, C, H, W) -> (B, N, embed_dim)
        # where N = (H*W)/(patch_size^2)
        pass
```

Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        # TODO: Implement multi-head attention
        # - Q, K, V linear projections
        # - Output projection
        # - Proper head dimension calculation
        pass

    def forward(self, x):
        # TODO: Implement forward pass
        # 1. Linear projections to Q, K, V
        # 2. Reshape for multi-head
        # 3. Scaled dot-product attention
        # 4. Concatenate heads
        # 5. Output projection
        pass
```

ViT Implementation Details

- Use `nn.Conv2d(kernel_size=patch_size, stride=patch_size)` for patch embedding
- Add learnable CLS token and position embeddings
- Scale attention by $\sqrt{d_k}$ for stability
- Use GELU activation in MLP blocks
- Apply LayerNorm before attention and MLP (pre-norm)

Section 4

Part 4: Comparative Analysis

Part 4: Comprehensive Comparative Analysis

4.1 Performance Benchmarking (25 min)

- Train ResNet-50 and ViT-Base on CIFAR-10/100
- Measure training convergence, final accuracy, inference speed
- Vary dataset size: 1K, 10K, full dataset
- Plot learning curves and analyze data efficiency

4.2 Computational Profiling (20 min)

- Memory usage: Peak GPU memory during training/inference
- FLOPs analysis: Theoretical and measured computational cost
- Latency benchmarking: Batch size 1 vs 32 vs 128
- Energy consumption measurement (if hardware supports)

4.3 Architecture Selection Framework (15 min)

- Create decision tree: Dataset size $\rightarrow$ Computational budget $\rightarrow$ Task requirements
- Practical deployment scenarios and recommendations
- Trade-off analysis: Accuracy vs Speed vs Memory vs Data efficiency

Benchmarking and Profiling Framework

Performance Measurement

```python
class ModelProfiler:
    def __init__(self, model, device):
        self.model = model
        self.device = device

    def measure_flops(self, input_size):
        # TODO: Implement FLOPs counting
        # Use torchprofile or similar
        pass

    def measure_memory(self, batch_size, input_size):
        # TODO: Implement memory profiling
        # Track GPU memory usage
        pass

    def measure_latency(self, input_tensor, num_runs=100):
        # TODO: Implement latency measurement
        # Average over multiple runs
        pass

    def benchmark_training(self, dataloader, epochs=5):
        # TODO: Implement training benchmark
        # Track convergence speed
        pass

```

Comparison Metrics

- **Accuracy**: Top-1 and Top-5 on test set
- **Training time**: Time to reach target accuracy
- **Memory**: Peak GPU memory usage
- **FLOPs**: Computational complexity
- **Latency**: Inference time per image
- **Data efficiency**: Performance vs dataset size

Lab Assessment and Deliverables

Coding Assessment (60%)

Implementation Quality:

- Correct mathematical implementation
- Clean, well-documented code
- Proper error handling and edge cases
- Efficient memory and computational usage

Architecture Components:

- ResNet blocks (basic & bottleneck)
- SE and CBAM attention modules
- ViT patch embedding and self-attention
- Proper integration and testing

Analysis Report (30%)

Required Sections:

1. ResNet depth scaling analysis
2. Attention mechanism effectiveness
3. CNN vs ViT performance comparison
4. Computational trade-off analysis
5. Architecture selection recommendations

Presentation Demo (10%)

5-minute presentation including:

- Live demonstration of implemented architectures
- Key findings from comparative analysis
- Most surprising/interesting result
- Practical deployment recommendation

Submission Details

Due Date: 1 week from lab session

Format:

- Single Jupyter notebook with all code
- Embedded markdown analysis sections
- All plots and visualizations included
- Runnable on standard GPU hardware

Bonus Points (5%): Creative extensions or optimizations beyond requirements

Getting Started

- Download template from course website
- Set up environment: `pip install -r requirements.txt`
- Start with Part 1: ResNet implementation
- Use provided test cases to verify implementations