Lab 3: Architecture Implementation Mastery

From ResNet to Vision Transformers - Hands-On

Prof. David Olivieri

Artificial Vision (VIAR25/26) - UVigo

September 22, 2025

Visión General del Laboratorio 3: Dominio de la Implementación de Arquitecturas

Objetivos de Aprendizaje

Al final de este laboratorio, serás capaz de:

- **Implementar** ResNet a partir de principios matemáticos con un análisis adecuado del flujo de gradientes
- **Diseñar** e integrar mecanismos de atención (SE, CBAM) en arquitecturas existentes
- **Construir** componentes de Vision Transformer desde cero con codificación posicional
- **Comparar** el rendimiento de CNN vs ViT en distintos regímenes de datos y restricciones computacionales
- **Analizar** compensaciones arquitectónicas y tomar decisiones de despliegue fundamentadas

Estructura del Laboratorio (4 horas)

Parte 1: Análisis Profundo de ResNet (90 min)

- Revisión de fundamentos matemáticos
- Implementación de bloques Básico vs Bottleneck
- Visualización del flujo de gradientes
- Experimentos de escalado en profundidad

Parte 2: Integración de Atención (90 min)

- Implementación del módulo SE
- Atención espacial + de canal con CBAM
- Evaluación de rendimiento (benchmarking)
- Visualización de la atención

Continuación...

Parte 3: Vision Transformer (60 min)

- Implementación de patch embedding
- Autoatención multi-cabeza
- Arquitectura ViT completa

Parte 4: Análisis Comparativo (60 min)

- Evaluación comparativa CNN vs ViT
- Perfilado computacional
- Marco de selección de arquitecturas

Herramientas & Entorno

- PyTorch 2.0+, torchvision
- Conjuntos de datos CIFAR-10/100
- GPU recomendada (Google Colab OK)
- Weights & Biases para el seguimiento de experimentos

Section 1

Parte 1: Análisis Profundo de ResNet

Parte 1: Análisis Detallado de la Implementación de ResNet

1.1 Revisión de Fundamentos Matemáticos (15 min)

- Derivar la formulación del aprendizaje residual: $H(x) = F(x) + x$
- Analizar el flujo de gradientes: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial H} \cdot (1 + \frac{\partial F}{\partial x})$
- Importancia de la identidad de mapeo y el problema de degradación

1.2 Implementación de Bloques (45 min)

- `BasicBlock`: convoluciones 3×3→3×3 con conexión de salto (skip connection)
- `BottleneckBlock`: 1×1→3×3→1×1 con reducción de canales 4×
- Comparación pre-activación vs post-activación
- Manejo adecuado de padding y stride para igualar dimensiones

1.3 Escalado de Arquitectura (30 min)

- Construir ResNet-18, 34, 50, 101 a partir de bloques componentes
- Medir tiempo de entrenamiento, uso de memoria y convergencia
- Graficar relación entre precisión y profundidad
- Analizar cuándo las redes más profundas ayudan vs perjudican

Implementación de ResNet: Componentes Clave

Estructura del Bloque Básico

```python
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
```

Estructura del Bloque Bottleneck

```python
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
```

Puntos Clave de Implementación

- Manejar el desajuste de dimensiones con convolución 1×1 en la conexión de salto
- Aplicar el stride únicamente en la primera convolución de cada bloque
- Colocación de BatchNorm: después de la convolución, antes de ReLU (ResNet post-activación)
- Factor de expansión de 4 en los bloques bottleneck

Section 2

Parte 2: Integración de Mecanismos de Atención

Parte 2: Integración de Mecanismos de Atención

2.1 Implementación del Módulo SE (30 min)

- Global average pooling: $z_c = \frac{1}{H \times W} \sum_{i,j} x_{i,j,c}$
- Red de excitación: $s = \sigma(W_2 \delta(W_1 z))$ con ratio de reducción
- Multiplicación por canal: $\tilde{x}_{i,j,c} = s_c \cdot x_{i,j,c}$
- Integración en bloques ResNet

2.2 Implementación de CBAM (30 min)

- Atención por canal: aplicar el mecanismo SE
- Atención espacial: $M_s = \sigma(\text{conv}_{77}([\text{AvgPool}(F); \text{MaxPool}(F)]))$
- Aplicación secuencial: $F'' = M_s(F') \odot F'$ donde $F' = M_c(F) \odot F$

2.3 Análisis de Rendimiento (30 min)

- Comparar ResNet vs ResNet+SE vs ResNet+CBAM
- Medir sobrecarga computacional y uso de memoria
- Visualizar mapas de atención usando Grad-CAM
- Analizar qué canales/regiones reciben mayor atención

Mecanismos de Atención: SE y CBAM

Módulo SE

```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

```

Módulo CBAM

```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
```

Consejos de Implementación

- Usar `AdaptiveAvgPool2d(1)` para el global pooling
- Ratio de reducción típico de 16 para módulos SE
- Para la atención espacial de CBAM, concatenar las características de avg y max pooling
- Visualizar mapas de atención con `torch.nn.functional.interpolate`

Section 3

Parte 3: Vision Transformer

Parte 3: Vision Transformer desde Cero

3.1 Capa de Embedding de Parches (20 min)

- Imagen a parches: $(H, W, C) \to (N, P^2 \cdot C)$ donde $N = \frac{H \times W}{P^2}$
- Proyección lineal: $\mathbf{x}_i \to \mathbf{e}_i = \mathbf{x}_i \mathbf{W}_e + \mathbf{b}_e$
- Añadir el token de clase al inicio: $[\mathbf{x}_{\text{cls}}, \mathbf{e}_1, \ldots, \mathbf{e}_N]$
- Adición de codificación posicional (aprendible vs sinusoidal)

3.2 Autoatención Multi-Cabeza (25 min)

- Proyecciones Query, Key, Value para cada cabeza
- Atención producto punto escalado: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
- Concatenación multi-cabeza y proyección de salida
- Visualización e interpretación de pesos de atención

3.3 Arquitectura ViT Completa (15 min)

- Bloque codificador Transformer: LayerNorm $\to$ MSA $\to$ Add $\to$ LayerNorm $\to$ MLP $\to$ Add
- Apilar múltiples capas codificadoras
- Cabeza de clasificación: extraer token [CLS] $\to$ capa lineal
- Exploración de hiperparámetros: tamaño de parche, dimensión de embedding, número de cabezas

Vision Transformer: Componentes Clave

Embedding de Parches

```python
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
```

Atención Multi-Cabeza

```python
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
```

Detalles de Implementación de ViT

- Usar `nn.Conv2d(kernel_size=patch_size, stride=patch_size)` para el embedding de parches
- Añadir token CLS aprendible y embeddings posicionales
- Escalar la atención por $\sqrt{d_k}$ para mayor estabilidad
- Usar activación GELU en los bloques MLP
- Aplicar LayerNorm antes de la atención y del MLP (pre-norm)

Section 4

Parte 4: Análisis Comparativo

Parte 4: Análisis Comparativo Integral

4.1 Evaluación de Rendimiento (25 min)

- Entrenar ResNet-50 y ViT-Base en CIFAR-10/100
- Medir convergencia del entrenamiento, precisión final, velocidad de inferencia
- Variar el tamaño del dataset: 1K, 10K, dataset completo
- Graficar curvas de aprendizaje y analizar eficiencia de datos

4.2 Perfilado Computacional (20 min)

- Uso de memoria: pico de memoria GPU durante entrenamiento/inferencia
- Análisis de FLOPs: coste computacional teórico y medido
- Evaluación de latencia: batch size 1 vs 32 vs 128
- Medición de consumo energético (si el hardware lo permite)

4.3 Marco de Selección de Arquitecturas (15 min)

- Crear árbol de decisión: tamaño de dataset $\rightarrow$ presupuesto computacional $\rightarrow$ requisitos de la tarea
- Escenarios de despliegue práctico y recomendaciones
- Análisis de compensaciones: Precisión vs Velocidad vs Memoria vs Eficiencia de datos

Marco de Benchmarking y Perfilado

Medición de Rendimiento

```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

```

Métricas de Comparación

- **Precisión**: Top-1 y Top-5 en el conjunto de prueba
- **Tiempo de entrenamiento**: tiempo hasta alcanzar la precisión objetivo
- **Memoria**: pico de uso de memoria GPU
- **FLOPs**: complejidad computacional
- **Latencia**: tiempo de inferencia por imagen
- **Eficiencia de datos**: rendimiento en función del tamaño del dataset

Evaluación del Laboratorio y Entregables

Evaluación de Código (60%)

Calidad de la Implementación:

- Implementación matemática correcta
- Código limpio y bien documentado
- Manejo adecuado de errores y casos límite
- Uso eficiente de memoria y cómputo

Componentes de Arquitectura:

- Bloques ResNet (básico & bottleneck)
- Módulos de atención SE y CBAM
- Embedding de parches y autoatención en ViT
- Integración y pruebas adecuadas

Informe de Análisis (30%)

Secciones Requeridas:

1. Análisis del escalado en profundidad de ResNet
2. Efectividad de los mecanismos de atención
3. Comparación de rendimiento CNN vs ViT
4. Análisis de compensaciones computacionales
5. Recomendaciones de selección de arquitectura

Presentación/Demo (10%)

Presentación de 5 minutos incluyendo:

- Demostración en vivo de las arquitecturas implementadas
- Principales hallazgos del análisis comparativo
- Resultado más sorprendente/interesante
- Recomendación práctica de despliegue

Detalles de Entrega

Fecha límite: 1 semana después de la sesión del laboratorio
Formato:

- Un único cuaderno Jupyter con todo el código
- Secciones de análisis en markdown embebidas
- Todas las gráficas y visualizaciones incluidas
- Ejecutable en hardware GPU estándar

Puntos Extra (5%): Extensiones creativas u optimizaciones más allá de los requisitos

Primeros Pasos

- Descargar la plantilla desde la web del curso
- Configurar el entorno: `pip install -r requirements.txt`
- Empezar con la Parte 1: implementación de ResNet
- Usar los casos de prueba proporcionados para verificar implementaciones