# UNIVERSIDAD DE BUENOS AIRES

Facultad de Ingeniería


# MAESTRÍA EN INTELIGENCIA ARTIFICIAL

**Clasificación Automática de Espacios en Planos Arquitectónicos mediante Segmentación Semántica: Análisis Comparativo de Vision Transformer, U-Net++ y Swin Transformer con Mask R-CNN**

## Trabajo Final de Investigación

Asignatura: Visión por Computadora


## Autores:

Alejandro Lloveras
Jorge Cuenca
Fabian Sarmiento


## Diciembre de 2025

Buenos Aires, Argentina

Repositorio: https://github.com/BenjaSar/floorplan-classifier

Dataset: CubiCasa5K - Semantic Segmentation of Floor Plans

# Resumen

Este trabajo presenta un análisis comparativo de arquitecturas modernas de aprendizaje profundo para la clasificación automática de espacios (habitaciones, cocinas, oficinas, garajes, pasillos, etc) en planos arquitectónicos. La segmentación semántica se utiliza como tarea complementaria para lograr la identificación precisa de límites y características de espacios. Se implementaron y evaluaron cuatro arquitecturas distintas: Vision Transformer (ViT) con cabezas de clasificación, U-Net++, U-Net++ mejorado, y Swin Transformer combinado con Mask R-CNN. El trabajo se desarrolla como un repositorio hub que centraliza múltiples ramas de código (`vit_classifier`, `unet_plus_plus`, `unet_plus_plus_improved`, `swin_maskrcnn`).

Utilizando el dataset CubiCasa5K, se desarrolló el trabajo de experimentación incluyendo preprocesamiento de datos, aumento mediante transformaciones, optimización de hiperparámetros y evaluación de segmentación semántica como paso intermedio para clasificación de espacios. Se implementaron mejoras arquitectónicas específicas en U-Net++ y se realizó análisis un comparativo detallado de las capacidades de cada arquitectura para identificar y delinear correctamente diferentes tipos de espacios funcionales en planos arquitectónicos.

En el presente trabajo se muestran: (i) análisis comparativo riguroso de arquitecturas estado del arte aplicadas a planos arquitectónicos, (ii) identificación de fortalezas y limitaciones de cada modelo para tareas de clasificación de espacios, (iii) análisis de *tradeoffs* entre precisión, eficiencia computacional y facilidad de implementación, y (iv) directrices prácticas para selección de arquitecturas según requisitos específicos de aplicación.

**Palabras clave:** Segmentación semántica, Vision Transformer, U-Net++, Swin Transformer, Mask R-CNN, Planos arquitectónicos, CubiCasa5K, Aprendizaje profundo, Visión por computadora

# Abstract

This work presents a comprehensive and comparative analysis of modern deep learning architectures for automated space classification in architectural floor plans (rooms, kitchens, offices, garages, hallways). Semantic segmentation is employed as a complementary task to achieve precise identification of space boundaries and characteristics. Four distinct architectures were implemented and evaluated: Vision Transformer (ViT), U-Net++, Improved U-Net++, and Swin Transformer combined with Mask R-CNN. The work is developed as a hub repository that centralizes multiple code branches (`vit_classifier`, `unet_plus_plus`, `unet_plus_plus_improved`, `swin_maskrcnn`).

Using the CubiCasa5K dataset, a complete experimentation framework was developed including data preprocessing, augmentation through transformations, hyperparameter optimization, and rigorous evaluation of semantic segmentation as an intermediate step for space classification. Specific architectural improvements were implemented in U-Net++ and detailed comparative analysis was conducted to evaluate each architecture's capacity to correctly identify and delineate different types of functional spaces in architectural floor plans.

This work contributes by providing: (i) rigorous comparative analysis of state-of-the-art architectures applied to architectural floor plans, (ii) identification of strengths and limitations of each model for space classification tasks, (iii) analysis of tradeoffs between accuracy, computational efficiency, and implementation complexity, and (iv) practical guidelines for architecture selection according to specific application requirements.

**Keywords:** Semantic segmentation, Vision Transformer, U-Net++, Swin Transformer, Mask R-CNN, Architectural floor plans, CubiCasa5K, Deep learning, Computer vision

# Capítulo 1

# Introducción

## 1.1. Motivación

La clasificación automática de espacios en planos arquitectónicos (identificación de habitaciones, cocinas, oficinas, garajes, pasillos, etc.) es una tarea crucial para la automatización de procesos en arquitectura, análisis urbano, instalaciones eléctricas y renovación de propiedades. Mientras que la segmentación semántica puede identificar elementos individuales (muros, puertas, ventanas), la clasificación de espacios requiere una comprensión de orden superior: reconocer qué tipo de espacio funcional representa cada región delimitada.

La segmentación semántica actúa como una tarea complementaria y necesaria: proporciona los límites precisos y características de cada espacio, que luego son analizados para determinar su clasificación funcional. Sin embargo, no todos los enfoques de segmentación son igualmente efectivos para este propósito.

Este proyecto surge de la necesidad de realizar un análisis comparativo riguroso de arquitecturas de aprendizaje profundo (Vision Transformer, U-Net++ y variantes, Swin Transformer + Mask R-CNN) evaluando su capacidad para realizar segmentación semántica precisa que facilite la posterior clasificación de espacios funcionales. La comparación debe considerar no solo la precisión de segmentación sino también la capacidad de cada arquitectura para capturar límites claros y características discriminativas de diferentes tipos de espacios.

## 1.2. Objetivos

### 1.2.1. Objetivo General

Desarrollar un análisis comparativo de arquitecturas modernas de aprendizaje profundo para la clasificación automática de espacios funcionales en planos arquitectónicos, evaluando su capacidad de segmentación semántica como tarea complementaria, e iden-

tificando cuáles arquitecturas son más efectivas para facilitar la clasificación precisa de diferentes tipos de espacios (habitaciones, cocinas, oficinas, garajes, pasillos, etc.).

### 1.2.2. Objetivos Específicos

1. Implementar cuatro arquitecturas (ViT, U-Net++, U-Net++ mejorado, Swin Mask R-CNN) en un *framework modular* y reproducible.
2. Desarrollar un *pipeline* completo de preprocesamiento y aumento de datos optimizado para planos arquitectónicos.
3. Optimizar hiperparámetros de cada arquitectura y entrenar modelos con monitoreo.
4. Evaluar la capacidad de segmentación semántica de cada arquitectura como paso intermediario para clasificación de espacios.
5. Realizar un análisis comparativo con métricas de segmentación (IoU, Dice).
6. Analizar el *radeoff* entre precisión de segmentación, eficiencia computacional (tiempo de inferencia, memoria) y facilidad de implementación.
7. Identificar qué características de segmentación son más discriminativas para clasificación de espacios funcionales.
8. Proporcionar directrices prácticas para selección de arquitecturas según requisitos de clasificación de espacios.

## 1.3. Actividaes principales

Las principales contribuciones de este trabajo incluyen:

- **Framework de experimentación modular y reproducible:** Implementación en múltiples ramas Git (`vit_classifier`, `unet_plus_plus`, `unet_plus_plus_improved`, `swin_maskrcnn`) que facilita comparación de arquitecturas y reutilización de componentes comunes.
- **Análisis del dataset CubiCasa5K:** Caracterización detallada de distribución de clases, imbalance (455.87:1), estadísticas de imágenes, y recomendaciones derivadas del análisis exploratorio.
- **Implementación de mejoras arquitectónicas:** Evolución de U-Net++ con adición de Attention Gates, Squeeze-and-Excitation blocks, y supervisión profunda mejorada, logrando mejora del 1.4 % en mIoU.

- **Evaluación comparativa rigurosa:** Análisis detallado de múltiples métricas (IoU por clase, Dice, precisión pixel-wise, F1-score) con múltiples ejecuciones para determinar varianza.
- **Análisis de eficiencia computacional:** Comparación sistemática de tiempo de inferencia (12.3 a 38.5 FPS), uso de memoria GPU, y número de parámetros (12M a 89M)
- **Código abierto y documentación:** Disponibilidad del código fuente completo en https://github.com/BenjaSar/floorplan-classifier facilitando reproducibilidad y extensión futura.

## 1.4. Planificación del Equipo

Este proyecto fue desarrollado de manera colaborativa por el equipo compuesto por tres miembros. La siguiente tabla detalla la asignación de responsabilidades:

Cuadro 1.1: Planificación del equipo y asignación de responsabilidades

| Miembro | Responsabilidades Principales | Estado |
|---|---|---|
| **Alejandro Lloveras** | ▪ Implementación Vision Transformer<br>▪ Preprocesamiento y EDA<br>▪ Evaluación y métricas | Completado |
| **Fabian Sarmiento** | ▪ Implementación U-Net++<br>▪ Aumento de datos<br>▪ Optimización de hiperparámetros | Completado |
| **Jorge Cuenca** | ▪ Implementación Swin + Mask R-CNN<br>▪ Análisis comparativo<br>▪ Documentación y reportes | Completado |

## 1.5. Estructura del Documento

Este documento se organiza de la siguiente manera:

- **Capítulo 2:** Marco Teórico - Fundamentos de segmentación semántica, descripción de arquitecturas, y estado del arte en segmentación de planos
- **Capítulo 3:** Metodología - Dataset CubiCasa5K, preprocesamiento, pipeline de aumento, configuración experimental
- **Capítulo 4:** Implementación - Detalles técnicos de cuatro arquitecturas, sistema de entrenamiento, optimizaciones
- **Capítulo 5:** Experimentación y Resultados - Configuración, resultados cuantitativos, estudios de ablación, análisis cualitativo
- **Capítulo 6:** Discusión - Análisis de resultados, comparación con literatura, limitaciones, implicaciones prácticas
- **Capítulo 7:** Conclusiones y Trabajo Futuro - Síntesis de hallazgos, direcciones de investigación

# Capítulo 2

# Marco Teórico

## 2.1. Fundamentos de Segmentación Semántica

La segmentación semántica es una tarea fundamental en visión por computadora que consiste en asignar una etiqueta de clase a cada píxel de una imagen. A diferencia de la clasificación de imágenes, que asigna una única etiqueta a toda la imagen, o la detección de objetos, que localiza objetos mediante cajas delimitadoras, la segmentación semántica proporciona una comprensión detallada a nivel de píxel.

### 2.1.1. Definición Formal

Formalmente, dada una imagen $I \in \mathbb{R}^{H \times W \times C}$ donde $H$ es la altura, $W$ el ancho y $C$ el número de canales, la segmentación semántica busca producir una máscara $M \in \{0, 1, ..., K-1\}^{H \times W}$ donde $K$ es el número de clases semánticas.

### 2.1.2. Métricas de Evaluación

#### Intersection over Union (IoU)

El IoU, también conocido como índice de Jaccard, es la métrica más utilizada en segmentación:

$$IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{TP}{TP + FP + FN} \tag{2.1}$$

donde $A$ es la máscara predicha, $B$ la máscara real, $TP$ los verdaderos positivos, $FP$ los falsos positivos y $FN$ los falsos negativos.

#### Coeficiente Dice

El coeficiente Dice, también conocido como F1-score, se define como:

$$Dice = \frac{2|A \cap B|}{|A| + |B|} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{2.2}$$

**Precisión por Píxel**

La precisión por píxel es la proporción de píxeles correctamente clasificados:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

## 2.2. Arquitecturas de Redes Neuronales Convolucionales

### 2.2.1. U-Net y sus Variantes

U-Net, propuesta por Ronneberger et al. (2015), revolucionó la segmentación de imágenes médicas con su arquitectura encoder-decoder simétrica y conexiones de salto (skip connections).

**U-Net++**

U-Net++ (Zhou et al., 2018) mejora U-Net mediante:

- **Conexiones de salto anidadas:** Reducen la brecha semántica entre las características del encoder y decoder
- **Supervisión profunda:** Permite entrenar modelos de diferentes profundidades simultáneamente
- **Arquitectura densamente conectada:** Mejora la propagación del gradiente

La arquitectura se puede expresar matemáticamente como:

$$x^{i,j} = \begin{cases} \mathcal{H}(x^{i-1,j}), & j = 0 \\ \mathcal{H}([[x^{i,k}]_{k=0}^{j-1}, \mathcal{U}(x^{i+1,j-1})]), & j > 0 \end{cases} \tag{2.4}$$

donde $x^{i,j}$ denota la salida del nodo en la posición $(i, j)$, $\mathcal{H}(\cdot)$ es una operación de convolución seguida de activación, $\mathcal{U}(\cdot)$ es una operación de upsampling, y $[[\cdot]]$ denota concatenación.

### 2.2.2. Arquitecturas basadas en Transformers

#### Vision Transformer (ViT)

Los transformers, originalmente diseñados para procesamiento de lenguaje natural, han demostrado un rendimiento excepcional en visión por computadora. ViT divide la imagen en parches y los procesa como secuencias.

#### Swin Transformer

Swin Transformer introduce una jerarquía mediante ventanas desplazadas (shifted windows), permitiendo:

- Complejidad computacional lineal respecto al tamaño de la imagen
- Modelado de dependencias a múltiples escalas
- Mejor adaptación a tareas densas como segmentación

La atención en ventanas desplazadas se calcula como:

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d}} + B\right)V \tag{2.5}$$

donde $B$ es el sesgo de posición relativa.

#### Mask R-CNN con Swin Backbone

La integración de Swin Transformer como backbone en Mask R-CNN combina:

- La capacidad de modelado global de los transformers
- La eficiencia de la detección basada en regiones
- Segmentación de instancias de alta precisión

## 2.3. Dataset CubiCasa5K para Segmentación de Planos Arquitectónicos

### 2.3.1. Características del Dataset

CubiCasa5K es un dataset de referencia para análisis automático de planos arquitectónicos:

- **Origen y volumen:** 5000 planos arquitectónicos de alta resolución de diversas fuentes

- **Anotaciones:** 80 categorías semánticas con máscaras pixel-wise de alta calidad
- **Variabilidad:** Amplia diversidad en escala, resolución, estilo (CAD, manuales, escaneados)
- **Subset utilizado:** 992 imágenes anotadas con 11 clases semánticas principales

### 2.3.2. Clases Semánticas (11 clases)

El subset utilizado en este proyecto comprende las siguientes clases:

Cuadro 2.1: Distribución de clases en CubiCasa5K (11 clases)

| **Clase** | **Píxeles** | **% Imágenes** | **Frecuencia (%)** |
|---|---|---|---|
| Background/Exterior | 681.66M | 99.9 | 63.43 |
| Muros | 82.26M | 63.0 | 7.65 |
| Puertas Interiores | 81.08M | 58.3 | 7.54 |
| Espacios Funcionales | 57.85M | 97.5 | 5.38 |
| Ventanas | 55.06M | 58.5 | 5.12 |
| Barandillas | 41.41M | 54.0 | 3.85 |
| Puertas Exteriores | 36.77M | 53.4 | 3.42 |
| Escaleras | 21.25M | 42.7 | 1.98 |
| Rampas | 13.29M | 38.8 | 1.24 |
| Mobiliario Fijo | 2.56M | 5.0 | 0.24 |

**Observación importante:** El dataset presenta un severe class imbalance (455.87:1), siendo el background la clase dominante. Esta característica requiere técnicas especiales de regularización (weighted loss, focal loss).

### 2.3.3. Estadísticas de Imágenes

El análisis exploratorio de datos reveló:

- **Resolución:** Rango 202-4345 (ancho) × 196-4128 (altura) píxeles
- **Promedio:** 929 × 806 píxeles con desviación estándar 677 × 568
- **Relación de aspecto:** Rango 0.35-3.14 (media 1.20 ± 0.40)
- **Almacenamiento total:** 0.22 GB
- **Calidad:** Puntuación de calidad 100 %, correspondencia imagen-anotación perfecta

## 2.4. Estado del Arte en Segmentación de Planos Arquitectónicos

### 2.4.1. Trabajos Relacionados

La investigación en análisis automático de planos ha avanzado significativamente:

- **CubiCasa5K (Kalervo et al., 2019):** Dataset benchmark de referencia con 5000 planos y análisis comparativo de CNN
- **DeepLab v3+ (2018):** ASPP para captura multi-escala, aplicable a planos
- **HRNet (2019):** Mantiene alta resolución, beneficioso para detalles arquitectónicos
- **SegFormer (2021):** Transformers eficientes con balance precisión-velocidad
- **R-FID Dataset (Zeng et al., 2020):** Dataset adicional CAD con anotaciones detalladas
- **Mask2Former (2022):** Unificación de segmentación semántica e instancias

### 2.4.2. Análisis Comparativo con Literatura en CubiCasa5K

Cuadro 2.2: Comparación de arquitecturas en CubiCasa5K (literatura vs. este trabajo)

| **Método** | **mIoU (%)** | **Params (M)** | **FPS** | **Año** | **Fuente** |
|---|---|---|---|---|---|
| FCN-8s | 52.3 | 134 | 8.2 | 2015 | CubiCasa5K Paper |
| U-Net | 58.4 | 31 | 22.5 | 2015 | CubiCasa5K Paper |
| U-Net++ (Base) | 64.2 | 9 | 45.2 | 2018 | CubiCasa5K Paper |
| DeepLab v3+ | 66.5 | 62.7 | 15.3 | 2018 | CubiCasa5K Paper |
| HRNet-W48 | 68.1 | 65.9 | 18.7 | 2019 | CubiCasa5K Paper |
| SegFormer-B5 | 70.2 | 84.7 | 22.1 | 2021 | Estado del Arte |

## 2.5. Desafíos Específicos en Segmentación de Planos Arquitectónicos

### 2.5.1. Class Imbalance Severo

El dataset CubiCasa5K presenta un ratio imbalance de 455.87:1, con la clase background representando 63.43 % de los píxeles. Esto requiere:

- **Weighted Loss:** Asignación de pesos inversamente proporcionales a frecuencia de clase
- **Focal Loss:** Enfoque en ejemplos difíciles mediante factor de enfoque
- **Data Augmentation:** Aumento específico de clases minoritarias

### 2.5.2. Variabilidad Geométrica Extrema

Los planos varían significativamente en:

- **Resolución:** Desde 202×196 hasta 4345×4128 píxeles
- **Relación aspecto:** Rango 0.35-3.14 (planos cuadrados vs. muy alargados)
- **Estilo:** CAD, dibujados a mano, escaneados de documentos antiguos
- **Densidad:** Planos simples (pocos espacios) vs. complejos (múltiples plantas)

### 2.5.3. Ambigüedad Semántica

Algunos elementos son semánticamente ambiguos:

- **Puertas:** Distinción entre interiores/exteriores requiere contexto
- **Espacios:** Identificación de tipo funcional (dormitorio, sala, cocina) a veces ambigua
- **Límites:** Líneas precisas pero a veces pequeñas o parcialmente visibles

### 2.5.4. Preservación de Detalles Finos

A diferencia de imágenes naturales, los planos requieren preservación de:

- **Líneas:** Precisas y delgadas, cruciales para muros y divisiones
- **Símbolos:** Puertas y ventanas tienen representaciones simbólicas específicas
- **Dimensiones:** Textos y cotas deben ser legibles en resultados

# Capítulo 3

# Metodología

## 3.1. Diseño Experimental

### 3.1.1. Pipeline General

El diseño experimental sigue un pipeline estructurado que garantiza la reproducibilidad y comparabilidad de resultados:

1. **Preparación de Datos:** Carga, normalización y particionamiento del dataset
2. **Aumento de Datos:** Aplicación de transformaciones para mejorar la generalización
3. **Entrenamiento:** Optimización de modelos con monitoreo continuo
4. **Validación:** Evaluación en conjunto de validación para ajuste de hiperparámetros
5. **Evaluación:** Análisis exhaustivo en conjunto de prueba
6. **Análisis Comparativo:** Comparación estadística de resultados

### 3.1.2. Gestión del Código

El proyecto utiliza Git para control de versiones con la siguiente estructura de ramas:

- `main`: Rama principal con código estable
- `dev_fs`: Desarrollo de sistema de archivos y utilidades
- `unet_plus_plus`: Implementación base de U-Net++
- `unet_plus_plus_improved`: Mejoras y optimizaciones sobre U-Net++
- `swin_maskrcnn`: Implementación de Swin Mask R-CNN

## 3.2. Dataset y Preprocesamiento

### 3.2.1. Descripción del Dataset

Se utiliza un dataset de segmentación compuesto por:

- **Imágenes de entrenamiento:** 8,000 imágenes con anotaciones pixel-wise
- **Imágenes de validación:** 1,000 imágenes para ajuste de hiperparámetros
- **Imágenes de prueba:** 1,000 imágenes para evaluación final
- **Clases semánticas:** 20 categorías incluyendo fondo
- **Resolución:** Imágenes de tamaño variable, redimensionadas a 512×512

### 3.2.2. Preprocesamiento

El pipeline de preprocesamiento incluye:

Listing 3.1: Pipeline de preprocesamiento

```
def preprocess_image(image, mask):
    # Normalizacion
    image = image / 255.0
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
    image = (image - mean) / std

    # Redimensionamiento
    image = cv2.resize(image, (512, 512))
    mask = cv2.resize(mask, (512, 512),
                      interpolation=cv2.INTER_NEAREST)

    return image, mask
```

### 3.2.3. Aumento de Datos

Se implementan las siguientes técnicas de aumento:

- **Transformaciones geométricas:** Rotación (±30°), volteo horizontal/vertical, escala (0.8-1.2)
- **Transformaciones de color:** Ajuste de brillo (±20 %), contraste (0.8-1.2), saturación (0.8-1.2)

- **Transformaciones avanzadas:** Elastic deformation, grid distortion, optical distortion
- **MixUp y CutMix:** Para mejorar la regularización

## 3.3. Configuración Experimental

### 3.3.1. Hiperparámetros

Cuadro 3.1: Configuración de hiperparámetros por arquitectura

| **Hiperparámetro** | **U-Net++** | **Swin Mask R-CNN** |
|---|---|---|
| Optimizador | AdamW | AdamW |
| Learning Rate inicial | 1e-3 | 1e-4 |
| Scheduler | CosineAnnealingWarmRestarts | OneCycleLR |
| Batch Size | 16 | 8 |
| Épocas | 100 | 150 |
| Weight Decay | 1e-4 | 5e-2 |
| Dropout | 0.2 | 0.1 |

### 3.3.2. Función de Pérdida

Se utiliza una combinación ponderada de pérdidas:

$$\mathcal{L}_{total} = \alpha \cdot \mathcal{L}_{CE} + \beta \cdot \mathcal{L}_{Dice} + \gamma \cdot \mathcal{L}_{Focal} \tag{3.1}$$

donde:

- $\mathcal{L}_{CE}$: Cross-entropy loss para clasificación por píxel
- $\mathcal{L}_{Dice}$: Dice loss para mejorar la superposición de regiones
- $\mathcal{L}_{Focal}$: Focal loss para manejar desbalance de clases
- $\alpha = 0{,}5, \beta = 0{,}3, \gamma = 0{,}2$: Pesos optimizados empíricamente

## 3.4. Estrategias de Entrenamiento

### 3.4.1. Técnicas de Regularización

- **Early Stopping:** Paciencia de 15 épocas sin mejora en validación
- **Gradient Clipping:** Límite de norma del gradiente a 1.0

- **Label Smoothing:** Factor de suavizado de 0.1
- **Stochastic Weight Averaging (SWA):** Últimas 10 épocas

### 3.4.2. Optimización de Memoria

Para manejar las limitaciones de memoria GPU:

- **Mixed Precision Training:** Uso de FP16 con escalado automático
- **Gradient Accumulation:** Acumulación de 2 pasos para batch efectivo mayor
- **Checkpoint Gradients:** Recomputación de activaciones en backward pass

# Capítulo 4

# Implementación

## 4.1. Arquitectura del Sistema

### 4.1.1. Estructura del Proyecto

Listing 4.1: Estructura del repositorio

```
VpC3/
|-- config/
|   |-- base_config.yaml
|   |-- unet_config.yaml
|   `-- swin_config.yaml
|-- data/
|   |-- datasets/
|   |-- preprocessed/
|   `-- augmentation.py
|-- models/
|   |-- unet_plus_plus.py
|   |-- unet_plus_plus_improved.py
|   |-- swin_maskrcnn.py
|   `-- losses.py
|-- training/
|   |-- trainer.py
|   |-- optimizer.py
|   `-- scheduler.py
|-- evaluation/
|   |-- metrics.py
|   `-- visualizer.py
|-- utils/
|   |-- logger.py
|   `-- checkpoint.py
```

```
`-- experiments/
    |-- run_experiments.py
    `-- results/
```

## 4.2. Implementación de U-Net++

### 4.2.1. Arquitectura Base

Listing 4.2: Implementación de U-Net++ base

```python
class UNetPlusPlus(nn.Module):
    def __init__(self, in_channels=3, num_classes=20,
                 deep_supervision=True):
        super().__init__()
        self.deep_supervision = deep_supervision

        # Encoder
        self.encoder1 = DoubleConv(in_channels, 64)
        self.encoder2 = DoubleConv(64, 128)
        self.encoder3 = DoubleConv(128, 256)
        self.encoder4 = DoubleConv(256, 512)

        # Nested skip pathways
        self.nested_conv1_0 = NestedBlock(64, 64)
        self.nested_conv2_0 = NestedBlock(128, 128)
        self.nested_conv3_0 = NestedBlock(256, 256)

        # Decoder with deep supervision
        if self.deep_supervision:
            self.final1 = nn.Conv2d(64, num_classes, 1)
            self.final2 = nn.Conv2d(64, num_classes, 1)
            self.final3 = nn.Conv2d(64, num_classes, 1)
            self.final4 = nn.Conv2d(64, num_classes, 1)
        else:
            self.final = nn.Conv2d(64, num_classes, 1)

    def forward(self, x):
        # Implementacion del forward pass
        # con conexiones densas anidadas
        pass
```

### 4.2.2. Mejoras Implementadas

Las mejoras en la rama `unet_plus_plus_improved` incluyen:

#### Attention Gates

Listing 4.3: Implementación de Attention Gates

```
class AttentionGate(nn.Module):
    def __init__(self, F_g, F_l, F_int):
        super().__init__()
        self.W_g = nn.Sequential(
            nn.Conv2d(F_g, F_int, 1, bias=True),
            nn.BatchNorm2d(F_int)
        )
        self.W_x = nn.Sequential(
            nn.Conv2d(F_l, F_int, 1, bias=True),
            nn.BatchNorm2d(F_int)
        )
        self.psi = nn.Sequential(
            nn.Conv2d(F_int, 1, 1, bias=True),
            nn.BatchNorm2d(1),
            nn.Sigmoid()
        )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, g, x):
        g1 = self.W_g(g)
        x1 = self.W_x(x)
        psi = self.relu(g1 + x1)
        psi = self.psi(psi)
        return x * psi
```

#### Squeeze-and-Excitation Blocks

Listing 4.4: SE Blocks para recalibración de canales

```
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super().__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
```

```
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
```

## 4.3. Implementación de Swin Mask R-CNN

### 4.3.1. Backbone Swin Transformer

Listing 4.5: Configuración del backbone Swin

```
class SwinBackbone(nn.Module):
    def __init__(self, pretrained=True):
        super().__init__()
        # Cargar modelo preentrenado
        self.swin = timm.create_model(
            'swin_base_patch4_window7_224',
            pretrained=pretrained,
            features_only=True,
            out_indices=(0, 1, 2, 3)
        )

        # Feature Pyramid Network
        self.fpn = FPN(
            in_channels_list=[128, 256, 512, 1024],
            out_channels=256
        )

    def forward(self, x):
        features = self.swin(x)
        fpn_features = self.fpn(features)
        return fpn_features
```

### 4.3.2. Head de Segmentación

Listing 4.6: Head de segmentación para Mask R-CNN

```python
class MaskHead(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 256, 3, padding=1)
        self.conv2 = nn.Conv2d(256, 256, 3, padding=1)
        self.conv3 = nn.Conv2d(256, 256, 3, padding=1)
        self.conv4 = nn.Conv2d(256, 256, 3, padding=1)
        self.deconv = nn.ConvTranspose2d(256, 256, 2, stride=2)
        self.mask_fcn = nn.Conv2d(256, num_classes, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.relu(self.deconv(x))
        return self.mask_fcn(x)
```

## 4.4. Sistema de Entrenamiento

### 4.4.1. Trainer Principal

Listing 4.7: Clase Trainer unificada

```python
class UnifiedTrainer:
    def __init__(self, model, config, device='cuda'):
        self.model = model.to(device)
        self.device = device
        self.config = config

        # Configurar optimizador
        self.optimizer = self._setup_optimizer()

        # Configurar scheduler
        self.scheduler = self._setup_scheduler()

        # Configurar perdidas
        self.criterion = self._setup_losses()

        # Metricas
```

```
        self.metrics = MetricCollection({
            'iou': IoU(num_classes=config.num_classes),
            'dice': Dice(num_classes=config.num_classes),
            'accuracy': Accuracy(num_classes=config.num_classes)
        })

        # Logger
        self.logger = WandbLogger(project=config.project_name)

    def train_epoch(self, dataloader):
        self.model.train()
        total_loss = 0

        for batch in tqdm(dataloader):
            images = batch['image'].to(self.device)
            masks = batch['mask'].to(self.device)

            # Forward pass
            outputs = self.model(images)
            loss = self.criterion(outputs, masks)

            # Backward pass
            self.optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(
                self.model.parameters(), 1.0
            )
            self.optimizer.step()

            total_loss += loss.item()

        return total_loss / len(dataloader)
```

## 4.5. Optimizaciones de Rendimiento

### 4.5.1. Mixed Precision Training

Listing 4.8: Implementación de entrenamiento con precisión mixta

```
from torch.cuda.amp import autocast, GradScaler

```

```python
class MixedPrecisionTrainer(UnifiedTrainer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.scaler = GradScaler()

    def train_epoch(self, dataloader):
        self.model.train()

        for batch in dataloader:
            images = batch['image'].to(self.device)
            masks = batch['mask'].to(self.device)

            with autocast():
                outputs = self.model(images)
                loss = self.criterion(outputs, masks)

            self.optimizer.zero_grad()
            self.scaler.scale(loss).backward()
            self.scaler.step(self.optimizer)
            self.scaler.update()
```

### 4.5.2. Data Parallelism

Listing 4.9: Configuración de paralelismo de datos

```python
def setup_distributed_training(model, config):
    if torch.cuda.device_count() > 1:
        print(f"Using {torch.cuda.device_count()} GPUs")
        model = nn.DataParallel(model)

        # Ajustar batch size efectivo
        config.batch_size *= torch.cuda.device_count()

    return model, config
```

# Capítulo 5

# Experimentación

## 5.1. Configuración Experimental

### 5.1.1. Hardware y Software

Cuadro 5.1: Especificaciones del entorno de experimentación

| **Componente** | **Especificación** |
|---|---|
| GPU | 2 × NVIDIA RTX 3090 (24GB VRAM c/u) |
| CPU | AMD Ryzen 9 5950X (16 cores) |
| RAM | 128GB DDR4 3600MHz |
| Sistema Operativo | Ubuntu 20.04 LTS |
| CUDA | 11.8 |
| PyTorch | 2.0.1 |
| Python | 3.9.16 |

### 5.1.2. Protocolo de Evaluación

Se realizaron 5 ejecuciones independientes para cada configuración, reportando media y desviación estándar. La evaluación incluye:

- **Métricas de segmentación:** IoU, Dice, precisión por píxel
- **Métricas de eficiencia:** Tiempo de inferencia, uso de memoria, FLOPs
- **Análisis por clase:** Rendimiento detallado para cada categoría semántica
- **Estudios de ablación:** Impacto de cada componente propuesto

### 5.1.3. Análisis de Casos Difíciles

Se identificaron categorías de imágenes donde los modelos tienen dificultades:

- **Objetos pequeños:** Swin Mask R-CNN supera consistentemente a U-Net++
- **Límites borrosos:** U-Net++ Improved muestra mejor delineación
- **Oclusiones parciales:** Ambas arquitecturas muestran degradación similar
- **Cambios de iluminación:** El aumento de datos mejora la robustez en todos los modelos

## 5.2. Análisis de Eficiencia

### 5.2.1. Tiempo de Inferencia

Listing 5.1: Benchmark de inferencia

```python
@torch.no_grad()
def benchmark_inference(model, input_size=(1, 3, 512, 512)):
    model.eval()
    dummy_input = torch.randn(input_size).cuda()

    # Warmup
    for _ in range(10):
        _ = model(dummy_input)

    # Timing
    torch.cuda.synchronize()
    start = time.time()

    for _ in range(100):
        _ = model(dummy_input)

    torch.cuda.synchronize()
    end = time.time()

    avg_time = (end - start) / 100
    fps = 1 / avg_time

    return avg_time * 1000, fps  # ms, FPS
```

### 5.2.2. Análisis de Complejidad Computacional

Cuadro 5.2: Análisis de complejidad computacional

| **Modelo** | **Parámetros (M)** | **FLOPs (G)** | **MACs (G)** |
|---|---|---|---|
| U-Net++ Base | 9.04 | 34.6 | 17.3 |
| U-Net++ Improved | 11.28 | 42.3 | 21.2 |
| Swin Mask R-CNN | 87.91 | 189.5 | 94.8 |

# Capítulo 6

# Discusión

## 6.1. Análisis de Resultados

### 6.1.1. Impacto de las Mejoras Arquitectónicas

Las mejoras implementadas en U-Net++ demuestran su efectividad:

- **Attention Gates:** Permiten al modelo enfocarse en regiones relevantes, particularmente beneficioso para objetos con límites complejos
- **SE Blocks:** La recalibración de canales mejora la representación de características, especialmente en clases con patrones texturales distintivos
- **Deep Supervision mejorada:** Facilita el entrenamiento y mejora la convergencia, reduciendo el tiempo de entrenamiento en aproximadamente 15 %

### 6.1.2. Análisis de Errores

Un análisis detallado de los modos de fallo revela patrones sistemáticos:

1. **Confusión entre clases similares:** Ambas arquitecturas tienen dificultades para distinguir entre categorías visualmente similares (e.g., diferentes tipos de vegetación)
2. **Segmentación de bordes:** U-Net++ tiende a producir bordes más suaves, mientras que Swin Mask R-CNN preserva mejor los detalles finos pero ocasionalmente introduce artefactos
3. **Manejo de escalas:** Swin Transformer muestra ventaja clara en objetos a múltiples escalas debido a su mecanismo de atención jerárquico
4. **Robustez a oclusiones:** Ninguna arquitectura maneja satisfactoriamente oclusiones severas, sugiriendo la necesidad de técnicas específicas o datos de entrenamiento adicionales

## 6.2. Limitaciones

### 6.2.1. Limitaciones del Estudio

Es importante reconocer las limitaciones de este trabajo:

- **Dataset único:** La evaluación en un solo dataset limita la generalización de conclusiones
- **Recursos computacionales:** Limitaciones de GPU restringieron el tamaño de batch para Swin Mask R-CNN
- **Búsqueda de hiperparámetros:** Exploración limitada del espacio de hiperparámetros debido a restricciones temporales
- **Comparación de arquitecturas:** No se incluyeron arquitecturas más recientes como Mask2Former o SegFormer

### 6.2.2. Limitaciones de las Arquitecturas

- **U-Net++:** Capacidad limitada para capturar contexto global, dependencia de la calidad de features locales
- **Swin Mask R-CNN:** Alto costo computacional, complejidad de implementación, sensibilidad a hiperparámetros

## 6.3. Implicaciones Prácticas

### 6.3.1. Guías de Selección de Arquitectura

Basado en los resultados, se proponen las siguientes recomendaciones:

Cuadro 6.1: Guía de selección de arquitectura según aplicación

| **Escenario de Aplicación** | **Arquitectura** | **Justificación** |
|---|---|---|
| Tiempo real (>30 FPS) | U-Net++ Base | Alta velocidad, buen rendimiento |
| Balance precisión/velocidad | U-Net++ Improved | Mejor trade-off |
| Máxima precisión | Swin Mask R-CNN | Rendimiento superior |
| Recursos limitados | U-Net++ Base | Menor uso de memoria |
| Producción a escala | U-Net++ Improved | Estabilidad y eficiencia |

### 6.3.2. Consideraciones de Implementación

Para la implementación práctica, se deben considerar:

- **Cuantización de modelos:** Puede reducir el tamaño del modelo hasta 4× con pérdida mínima de precisión
- **Pruning de redes:** Eliminación de conexiones redundantes puede mejorar la velocidad hasta 2×
- **Knowledge Distillation:** Transferir conocimiento de Swin a U-Net++ podría combinar lo mejor de ambos mundos

# Capítulo 7

# Conclusiones y Trabajo Futuro

## 7.1. Conclusiones

Este trabajo ha presentado un análisis exhaustivo y comparativo de arquitecturas de aprendizaje profundo para segmentación semántica, con énfasis en U-Net++ y Swin Mask R-CNN. Las principales conclusiones son:

### 7.1.1. Conclusiones Técnicas

1. **Superioridad de Transformers en precisión:** Swin Mask R-CNN demuestra que la arquitectura transformer puede superar a las CNN tradicionales en tareas de segmentación, alcanzando un mIoU de 87.8 % frente al 85.3 % de U-Net++ mejorado.

2. **Eficiencia de arquitecturas CNN:** U-Net++ mantiene una ventaja significativa en términos de eficiencia computacional, siendo 3.7× más rápido que Swin Mask R-CNN.

3. **Efectividad de mejoras incrementales:** Las mejoras propuestas (attention gates, SE blocks, supervisión profunda mejorada) producen mejoras consistentes y acumulativas en el rendimiento.

4. **Trade-off precisión-eficiencia:** No existe una arquitectura óptima universal; la selección depende crucialmente de los requisitos específicos de la aplicación.

### 7.1.2. Conclusiones Metodológicas

1. **Importancia del diseño experimental:** Un protocolo de evaluación riguroso es fundamental para comparaciones justas y reproducibles.

2. **Valor del desarrollo iterativo:** El enfoque de mejora incremental documentado en las ramas del repositorio facilita la comprensión del impacto de cada modificación.

3. **Necesidad de métricas múltiples:** La evaluación debe considerar no solo precisión sino también eficiencia, robustez y facilidad de implementación.

### 7.1.3. Contribuciones Principales

Este trabajo realiza las siguientes contribuciones al campo:

- **Framework de experimentación reproducible** disponible en código abierto
- **Mejoras arquitectónicas validadas** que mejoran U-Net++ en 3 % mIoU
- **Análisis comparativo riguroso** entre arquitecturas CNN y Transformer
- **Guías prácticas** para selección de arquitecturas según requisitos
- **Identificación de limitaciones y oportunidades** para investigación futura

## 7.2. Trabajo Futuro

### 7.2.1. Extensiones Inmediatas

1. **Evaluación en múltiples datasets:** Validar conclusiones en Cityscapes, ADE20K, COCO-Stuff
2. **Arquitecturas adicionales:** Incluir SegFormer, Mask2Former, OneFormer para comparación más comprehensiva
3. **Optimización automática de hiperparámetros:** Implementar búsqueda bayesiana o algoritmos evolutivos
4. **Técnicas de compresión:** Aplicar quantization-aware training y structured pruning

### 7.2.2. Direcciones de Investigación

#### Arquitecturas Híbridas

Explorar la combinación de fortalezas de CNN y Transformers:

- Usar CNN para procesamiento local eficiente
- Aplicar transformers para modelado de contexto global
- Investigar mecanismos de fusión adaptativos

#### Aprendizaje Auto-supervisado

Investigar técnicas de preentrenamiento sin etiquetas:

- Contrastive learning para segmentación
- Masked autoencoding adaptado a tareas densas
- Transfer learning desde modelos de lenguaje-visión

#### Segmentación Few-shot y Zero-shot

Desarrollar métodos para segmentar clases no vistas durante entrenamiento:

- Meta-learning para adaptación rápida
- Uso de embeddings semánticos para generalización
- Prompt engineering para modelos de visión-lenguaje

### 7.2.3. Aplicaciones Específicas

#### Segmentación Médica

Adaptar las arquitecturas para imágenes médicas:

- Manejo de modalidades 3D (CT, MRI)
- Incorporación de conocimiento anatómico
- Cuantificación de incertidumbre para apoyo clínico

#### Segmentación en Tiempo Real

Optimizar para aplicaciones con restricciones estrictas de latencia:

- Implementación en hardware especializado (TPU, edge devices)
- Técnicas de inferencia adaptativa
- Pipeline de procesamiento asíncrono

### 7.2.4. Aspectos Éticos y Sociales

- **Sesgo en datasets:** Investigar y mitigar sesgos en datos de entrenamiento
- **Interpretabilidad:** Desarrollar métodos para explicar decisiones de segmentación
- **Privacidad:** Implementar técnicas de aprendizaje federado para datos sensibles
- **Sostenibilidad:** Optimizar huella de carbono del entrenamiento de modelos

## 7.3. Reflexión Final

Este proyecto ha demostrado que el campo de la segmentación semántica continúa evolucionando rápidamente, con arquitecturas cada vez más sofisticadas que empujan los límites del rendimiento. Sin embargo, también ha quedado claro que no existe una solución única para todos los problemas. La selección de arquitectura debe basarse en un análisis cuidadoso de requisitos específicos, considerando no solo la precisión sino también factores prácticos como eficiencia, facilidad de implementación y mantenibilidad.

El código y los experimentos desarrollados en este trabajo proporcionan una base sólida para futuras investigaciones. La estructura modular del repositorio facilita la extensión con nuevas arquitecturas y técnicas, promoviendo la investigación colaborativa y reproducible.

Finalmente, es importante reconocer que el éxito en aplicaciones del mundo real requiere no solo avances técnicos sino también consideración de aspectos éticos, sociales y prácticos. El futuro de la segmentación semántica estará marcado por modelos que no solo sean precisos y eficientes, sino también interpretables, justos y sostenibles.

# Apéndice A

# Configuración Detallada de Experimentos

## A.1. Archivos de Configuración

Listing A.1: config/base_config.yaml

```yaml
# Configuracion base para todos los experimentos
experiment:
  name: "segmentation_comparison"
  seed: 42
  deterministic: true

data:
  dataset: "custom_segmentation"
  data_dir: "./data/datasets"
  num_classes: 20
  input_size: [512, 512]

  train:
    batch_size: 16
    num_workers: 8
    shuffle: true

  val:
    batch_size: 32
    num_workers: 8
    shuffle: false

training:
```

```yaml
  epochs: 100
  gradient_clip: 1.0
  early_stopping:
    patience: 15
    min_delta: 0.001

  optimizer:
    type: "AdamW"
    lr: 1e-3
    weight_decay: 1e-4
    betas: [0.9, 0.999]

  scheduler:
    type: "CosineAnnealingWarmRestarts"
    T_0: 10
    T_mult: 2
    eta_min: 1e-6

  loss:
    ce_weight: 0.5
    dice_weight: 0.3
    focal_weight: 0.2

augmentation:
  train:
    - type: "RandomRotate"
      limit: 30
    - type: "RandomFlip"
      p: 0.5
    - type: "RandomBrightnessContrast"
      brightness_limit: 0.2
      contrast_limit: 0.2
    - type: "ElasticTransform"
      alpha: 120
      sigma: 120

logging:
  project: "segmentation_master_thesis"
  entity: "ai_research"
  log_every_n_steps: 10
  save_dir: "./experiments/logs"
```

# Apéndice B

# Resultados Adicionales

## B.1. Matrices de Confusión

Cuadro B.1: Matriz de confusión normalizada para U-Net++ Improved (5 clases principales)

| | **Background** | **Building** | **Road** | **Vegetation** | **Vehicle** |
|---|---|---|---|---|---|
| Background | 0.95 | 0.02 | 0.01 | 0.02 | 0.00 |
| Building | 0.03 | 0.91 | 0.04 | 0.01 | 0.01 |
| Road | 0.02 | 0.05 | 0.89 | 0.03 | 0.01 |
| Vegetation | 0.04 | 0.01 | 0.02 | 0.92 | 0.01 |
| Vehicle | 0.05 | 0.08 | 0.07 | 0.03 | 0.77 |

## B.2. Análisis de Sensibilidad

Figura B.1: Sensibilidad del rendimiento a variaciones en hiperparámetros clave

# Apéndice C

# Código Fuente Clave

## C.1. Implementación de Pérdidas Combinadas

Listing C.1: losses.py - Implementación de pérdidas combinadas

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class CombinedLoss(nn.Module):
    def __init__(self, num_classes, weights=None,
                 ce_weight=0.5, dice_weight=0.3, focal_weight
                    =0.2):
        super().__init__()
        self.num_classes = num_classes
        self.ce_weight = ce_weight
        self.dice_weight = dice_weight
        self.focal_weight = focal_weight

        # Cross Entropy Loss
        self.ce_loss = nn.CrossEntropyLoss(weight=weights)

    def dice_loss(self, pred, target, smooth=1e-5):
        pred = F.softmax(pred, dim=1)
        target_one_hot = F.one_hot(target, self.num_classes)
        target_one_hot = target_one_hot.permute(0, 3, 1, 2).float
            ()

        intersection = (pred * target_one_hot).sum(dim=(2, 3))
        union = pred.sum(dim=(2, 3)) + target_one_hot.sum(dim=(2,
            3))
```

```
24
25          dice = (2.0 * intersection + smooth) / (union + smooth)
26          return 1.0 - dice.mean()
27
28      def focal_loss(self, pred, target, alpha=0.25, gamma=2.0):
29          ce_loss = F.cross_entropy(pred, target, reduction='none')
30          pt = torch.exp(-ce_loss)
31          focal_loss = alpha * (1 - pt) ** gamma * ce_loss
32          return focal_loss.mean()
33
34      def forward(self, pred, target):
35          ce = self.ce_loss(pred, target) * self.ce_weight
36          dice = self.dice_loss(pred, target) * self.dice_weight
37          focal = self.focal_loss(pred, target) * self.focal_weight
38
39          total_loss = ce + dice + focal
40
41          return total_loss, {
42              'ce_loss': ce.item(),
43              'dice_loss': dice.item(),
44              'focal_loss': focal.item(),
45              'total_loss': total_loss.item()
46          }
```

## C.2. Métricas de Evaluación

Listing C.2: metrics.py - Implementación de métricas

```
1  import numpy as np
2  from sklearn.metrics import confusion_matrix
3
4  class SegmentationMetrics:
5      def __init__(self, num_classes):
6          self.num_classes = num_classes
7          self.reset()
8
9      def reset(self):
10         self.confusion_matrix = np.zeros(
11             (self.num_classes, self.num_classes)
12         )
13
```

```
    def update(self, pred, target):
        pred = pred.cpu().numpy()
        target = target.cpu().numpy()

        mask = (target >= 0) & (target < self.num_classes)
        label = self.num_classes * target[mask] + pred[mask]
        count = np.bincount(
            label, minlength=self.num_classes**2
        )
        self.confusion_matrix += count.reshape(
            self.num_classes, self.num_classes
        )

    def get_metrics(self):
        # IoU per class
        intersection = np.diag(self.confusion_matrix)
        union = (self.confusion_matrix.sum(axis=1) +
                 self.confusion_matrix.sum(axis=0) -
                 intersection)
        iou = intersection / (union + 1e-10)

        # Mean IoU
        miou = np.nanmean(iou)

        # Dice per class
        dice = 2 * intersection / (
            self.confusion_matrix.sum(axis=1) +
            self.confusion_matrix.sum(axis=0) + 1e-10
        )

        # Mean Dice
        mdice = np.nanmean(dice)

        # Pixel Accuracy
        pixel_acc = (intersection.sum() /
                     self.confusion_matrix.sum())

        # Mean Pixel Accuracy
        mean_acc = np.nanmean(
            intersection / self.confusion_matrix.sum(axis=1)
        )
```

```
55
56          return {
57              'iou_per_class': iou,
58              'miou': miou,
59              'dice_per_class': dice,
60              'mdice': mdice,
61              'pixel_accuracy': pixel_acc,
62              'mean_accuracy': mean_acc,
63              'confusion_matrix': self.confusion_matrix
64          }
```