Practical Work 4: Advanced Vision, Segmentation, and 3D Data

Louis Fippo Fitime, Claude Tinku, Kerolle Sonfack
Department of Computer Engineering, ENSPY

October 8, 2025

Abstract

This fourth Practical Work focuses on advanced computer vision tasks that demand high precision and architectural complexity. Students will apply Deep Learning Engineering best practices to implement and evaluate a cutting-edge **U-Net architecture** for **semantic segmentation of medical images**. Furthermore, the module introduces the theoretical and practical challenges of handling **3D volumetric data** using Conv3D layers.

Learning Objectives

- Master the **semantic segmentation** task and the use of the **U-Net** architecture.
- Understand and implement specific metrics for segmentation (**IoU**, **Dice Coefficient**).
- Systematically apply **Deep Learning Engineering** best practices (experiment tracking, model packaging).
- Understand the core concepts and implementation of **3D Convolutions** for volumetric data.
- Recognize the specific challenges of medical image data (imbalance, limited size).

1 Part 1: Segmentation and MLOps Best Practices (1.5h)

1.1 Semantic Segmentation and the U-Net Architecture

Semantic segmentation is the task of assigning a class label to every pixel in an image, which is crucial in fields like autonomous driving and medicine.

- **Output Type:** Unlike classification, what is the dimension and nature of the output tensor of a semantic segmentation model?
- **U-Net Structure:** The U-Net is famous for its "U" shape and its use of **skip connections** between the contracting (encoder) and expansive (decoder) paths. Explain the role of the decoder path and how skip connections differ here compared to standard ResNet blocks.
- **Loss Functions:** Why is the standard categorical cross-entropy often inadequate for medical segmentation tasks where the foreground (e.g., tumor) is tiny compared to the background? Propose an alternative loss function.

1.2 Engineering Practices: Experiment Tracking

In advanced DL, reliable comparison between experiments is mandatory. We will continue to use MLflow (introduced in TP 2).

Instructions:

1. Define a strict naming convention for your MLflow runs to clearly identify the architecture, optimizer, and loss function used for each segmentation model trained.
2. Review the process of logging **custom metrics** (like the Dice Coefficient) which are not native to Keras's basic metrics list.

2 Part 2: Semantic Segmentation on Medical Data (3h)

2.1 Exercise 1: Implementing the U-Net Architecture

You will build a simplified 2D U-Net for segmenting a common medical dataset (e.g., a simulated cell or organ mask task). Assume the data has been pre-processed into normalized image arrays X and corresponding binary mask arrays Y.

Instructions:

1. Create a function for the core **convolutional block** (Conv-Batch Norm-ReLU).
2. Implement the **U-Net Encoder Path** using Max Pooling for downsampling.
3. Implement the **U-Net Decoder Path** using Conv2D Transpose (Upsampling).
4. **Crucial Step:** Implement the skip connections (concatenation) between the encoder and decoder.

```
import tensorflow as tf
from tensorflow import keras

def conv_block(input_tensor, num_filters):
# Core convolutional block
x = keras.layers.Conv2D(num_filters, (3, 3), padding='same')(input_tensor)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)

x = keras.layers.Conv2D(num_filters, (3, 3), padding='same')(x)
x = keras.layers.BatchNormalization()(x)
x = keras.layers.Activation('relu')(x)
return x

def build_unet(input_shape=(128, 128, 1)):
inputs = keras.Input(input_shape)

# ENCODER PATH (Contracting)
c1 = conv_block(inputs, 32)
p1 = keras.layers.MaxPooling2D((2, 2))(c1)

# TODO: Implement 2 more contracting steps (c2, p2 and c3, p3)
# The number of filters should double at each step (64, 128)

# BRIDGE / BOTTLENECK (256 filters)
b = conv_block(p3, 256)

# DECODER PATH (Expansive)
# Step 1: Upsampling + Skip Connection
u1 = keras.layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(b)
# TODO: Concatenate the corresponding encoder output (c3) with u1
```


Louis Fippo Fitime: louis.fippo@univ-yaounde1.cm

```
u1 = keras.layers.Concatenate()([u1, c3])
d1 = conv_block(u1, 128)

# TODO: Implement 2 more expansive steps (u2, d2 and u3, d3)
# ...

# Output Layer: 1 filter (for binary segmentation) with sigmoid
    activation
outputs = keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(d3)

return keras.Model(inputs=[inputs], outputs=[outputs])

TODO: Compile and train the model using a suitable loss and custom
    metric (IoU/Dice)
model.compile(...)
```

Listing 1: Simplified 2D U-Net Implementation with Keras

2.2 Exercise 2: Segmentation-Specific Metrics

The Dice Similarity Coefficient (or F1-Score for segmentation) is a common metric. It is defined as:

$$\mathrm{Dice} = \frac{2 \cdot |\mathrm{A} \cap \mathrm{B}|}{|\mathrm{A}| + |\mathrm{B}|}$$

where A is the predicted mask and B is the ground truth mask.

Instructions:

1. Implement the Dice Coefficient as a custom Keras metric function.
2. Implement the IoU (Intersection over Union) metric.
3. **Question:** Compare the sensitivity of the IoU metric versus the Dice Coefficient to small segmentation errors, especially when the target mask is small.

```
from keras import backend as K

def dice_coeff(y_true, y_pred, smooth=1.):
# Flatten the tensors for calculation
y_true_f = K.flatten(y_true)
y_pred_f = K.flatten(y_pred)

# Calculate intersection and union
intersection = K.sum(y_true_f * y_pred_f)

# TODO: Complete the Dice formula calculation
# return ...

# Example for IoU
def iou_metric(y_true, y_pred):
    intersection = K.sum(K.abs(y_true * y_pred), axis=-1)
```


Louis Fippo Fitime: louis.fippo@univ-yaounde1.cm

```
    union = K.sum(y_true, axis=-1) + K.sum(y_pred, axis=-1) -
   intersection
    return K.mean((intersection + smooth) / (union + smooth), axis
   =-1)
```

Listing 2: Dice Coefficient Metric Implementation

3 Part 3: Introduction to 3D Convolutions and Volumetric Data (2.5h)

3.1 Concepts: Conv3D for Volumetric Data

Medical imaging often involves volumetric data (e.g., CT scans, MRI) which are stacks of 2D slices, forming a D×H×W×C tensor (Depth, Height, Width, Channels). Standard Conv2D layers only process H×W spatial dimensions, ignoring the D dimension.

- **Conv3D Operation:** Describe how a Conv3D kernel differs from a Conv2D kernel in terms of dimensions and movement. Why is this necessary for volumetric data?
- **Memory Challenge:** Conv3D layers are computationally expensive. What engineering trade-offs must be made regarding kernel size, number of filters, or input depth (D) when designing a Conv3D architecture?

3.2 Exercise 3: Conv3D Block and Engineering Discipline

Instructions:

1. Define a simple Conv3D block in Keras to understand the input and output shapes.
2. Integrate **MLflow tracking** to log the full model configuration and key metrics for the Conv3D block experiment.

```
import mlflow
import numpy as np

def simple_conv3d_block(input_shape=(32, 32, 32, 1)):
# Simple block for demonstration: D x H x W x C
inputs = keras.Input(input_shape)

# Conv3D layer: 16 filters, 3x3x3 kernel
x = keras.layers.Conv3D(16, (3, 3, 3), activation='relu', padding='
    same')(inputs)
x = keras.layers.MaxPool3D((2, 2, 2))(x)

# TODO: Add a second Conv3D block (32 filters) and another MaxPool3D

x = keras.layers.Flatten()(x)
outputs = keras.layers.Dense(1, activation='sigmoid')(x) # Dummy
    output
return keras.Model(inputs, outputs)

if name == 'main':
mlflow.set_experiment("3D_Volumetric_Analysis")
with mlflow.start_run(run_name="Conv3D_Baseline"):
model_3d = simple_conv3d_block()

    # Log Architecture (Engineering Practice)
```

```
    model_config = model_3d.to_json()
    mlflow.log_dict({"model_config": model_config}, "artifacts/
model_architecture.json")

    # Log Hyperparameters
    mlflow.log_param("optimizer", "adam")
    mlflow.log_param("filters_start", 16)

    # Simulate training and log metrics
    # TODO: Simulate training by logging a final metric value (e.g.,
    final_val_loss)
    # mlflow.log_metric(...)
    print("MLflow tracking complete for 3D block experiment.")
```

Listing 3: Simple Conv3D Block with MLOps Logging

4 Conclusion

To submit:

- The link to your GitHub repository with the complete and executable `unet_segmentation.py` file, including the Dice and IoU metrics.
- A short report (`.pdf` or Overleaf link) detailing the results of your segmentation experiment, comparing the convergence speed and final performance (Dice and IoU) of your U-Net model with the concepts introduced in Part 3.

Louis Fippo Fitime: louis.fippo@univ-yaounde1.cm