Machine Learning for Statistical NLP: Advanced

Year: 2025/2026

Assignment 1

Chinese Characters segmentation

3rd October 2025

Andrea De Biasi

Index

1 Introduction

1.1 Project Goal and Task Description

This project addresses a fundamental task in computer vision: **Chinese characters localization** within images. Unlike traditional object detection, which returns coordinates for bounding boxes, our task is formulated as a **semantic segmentation problem**.
The primary goal is to develop a deep learning model that takes a raw image as input and outputs a **"soft binary" map** (a probability map). In this map, each pixel's value (ranging from 0 to 1) represents the probability of that pixel belonging to a character's bounding box.

The solution of the pixel-wise classification approach is implemented using **PyTorch** and leverages GPU acceleration for training.

1.2 The Dataset

The data source for this project is a subset (of aroung 1000 images) of the **CTW dataset**. This dataset provides images of complex scenes (often street views) containing text, along with corresponding annotations.

The original task of Optical Character Recognition (OCR) is broken down into this segmentation task:

- **Input**: RGB images (with an original size of $2048 \times 2048 \times 3$).
- **Target (Objective)**: A binary mask (ground truth) where a pixel is set to 1 if it falls within a Chinese character's bounding box, and 0 otherwise.

During data preparation, the provided JSON annotations (which contain bounding box coordinates) were converted into these dense, pixel-level binary masks to train the segmentation models. Preliminary data analysis indicated a significant class imbalance, as the bounding box pixels occupy a very small area relative to the overall image (background).

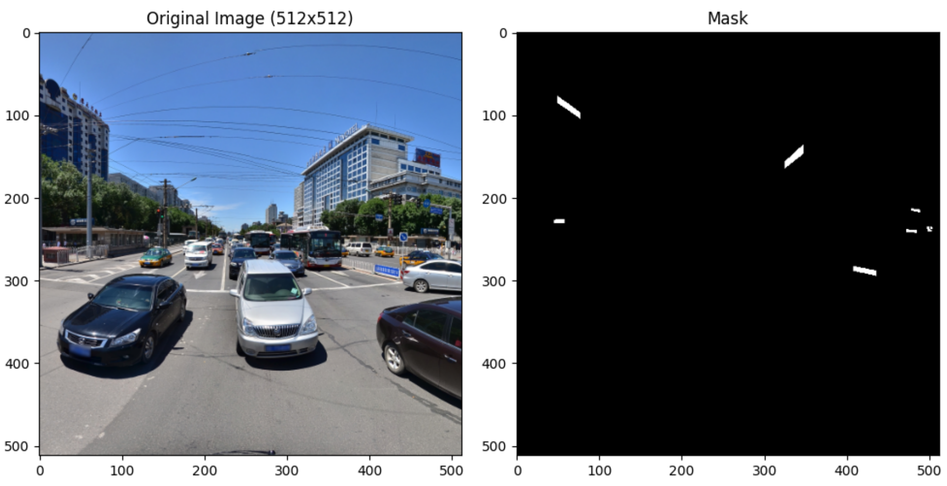

Figure 1: Example of an original image and its corresponding ground truth mask. Note the small area occupied by the target bounding boxes (white pixels) relative to the image size, highlighting the class imbalance problem.

2 Model Architectures

As required by the assignment, two substantially different model architectures were designed and implemented for this pixel-wise classification task. Both models follow the general **Encoder-Decoder** paradigm common in semantic segmentation.

2.1 Model 1: SimpleSegNet (SegNet-Inspired)

The first model, **SimpleSegNet**, is a streamlined version inspired by the **SegNet** architecture. This model is designed to efficiently map high-resolution input images to a dense, pixel-wise prediction map.

2.1.1 Architecture and Motivation

The core feature leveraged from SegNet is the use of **Max-Pooling Indices** for upsampling. During the encoder phase, the coordinates of the maximum value within each pooling window are stored. These indices are then transferred to the corresponding decoder layer to guide the **Max-Unpooling** operation.

◇ **Motivation**: This index-based unpooling method is crucial because it helps to maintain spatial information and sharp boundaries of the predicted masks, which is essential for accurately localizing the small character bounding boxes.

2.1.2 Structural Overview of SimpleSegNet

Our implementation of **SimpleSegNet** uses a structure with **four Encoder blocks** and **four corresponding Decoder blocks**. This design reduces the input resolution by a factor of 8 (2^3 pooling operations in the encoder) before reaching the final bottleneck, offering a compromise between deep feature extraction and maintaining context. The channel sizes are intentionally reduced compared to the full SegNet to manage memory usage while processing large inputs (512×512 images):

1. **Encoder Blocks**: Each block consists of two sequential **Convolution-BatchNorm-ReLU** layers, followed by a 2×2 **Max Pooling** layer which returns the pooling indices. The channel progression is: Input $\rightarrow 32 \rightarrow 64 \rightarrow 128$.
2. **Bottleneck**: The last encoder block processes the 128-channel features without a final pooling step, acting as the bridge between the encoder and decoder.
3. **Decoder Blocks**: Each block starts with a **Max-Unpooling** operation, using the indices from its symmetrical encoder block to precisely upsample the feature map. This is followed by two sequential **Convolution-BatchNorm-ReLU** layers. The channel progression is $128 \rightarrow 64 \rightarrow 32 \rightarrow 16$.
4. **Output Layer**: The final layer is a 3×3 Convolution that maps the 16 features to the single desired output channel (1), representing the probability map.

The use of **sequential convolutions** within each block (e.g., in `_make_encoder_block`) allows the model to capture more complex features at the same spatial resolution before downsampling.

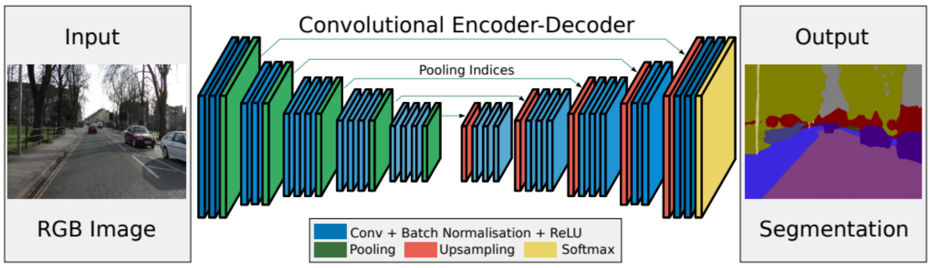

Figure 2: Conceptual diagram of the original SegNet Encoder-Decoder architecture

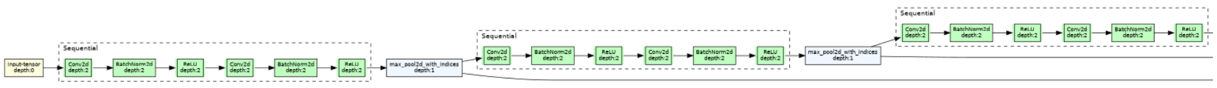

Figure 3: Diagram of my architecture: SimpleSegNet (encoder)

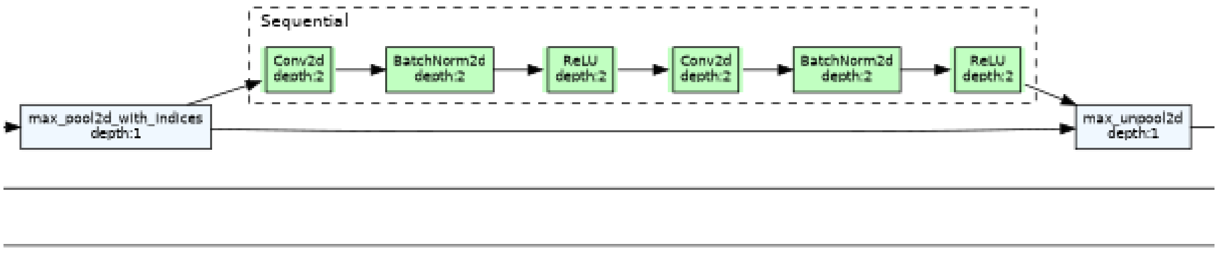

Figure 4: Diagram of my architecture: SimpleSegNet (bottleneck)

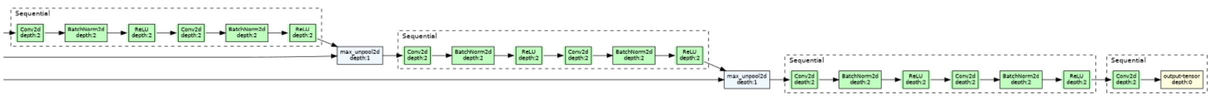

Figure 5: Diagram of my architecture: SimpleSegNet (decoder)

2.2 Model 2: CustomUNet (U-Net-Inspired)

The second model implemented is a deep convolutional network based on the **U-Net** architecture. U-Net is highly suitable for segmentation tasks, particularly in scenarios where detailed spatial information needs to be recovered, thanks to its extensive use of skip connections.

2.2.1 Architecture and Key Differences from SegNet

The primary architectural distinction between **CustomUNet** and **SimpleSegNet** lies in the method of information transfer from the encoder to the decoder:

- ◇ **Skip Connections (Concatenation)**: Instead of using pooling indices, U-Net transfers the feature maps from the corresponding encoder blocks directly to the decoder. These feature maps are **concatenated** along the channel dimension with the upsampled features from the previous decoder stage.
- ◇ **Upsampling**: The model uses a **Transposed Convolution** (`nn.ConvTranspose2d`) instead of Max-Unpooling. This method learns the upsampling parameters, potentially leading to smoother and more flexible feature map reconstruction.
- ◇ **Depth and Feature Channels**: **CustomUNet** is deeper, featuring **five levels** of downsampling/upsampling and utilizes a richer set of feature channels, culminating in 512 channels in the bottleneck.

2.2.2 Addressing Overfitting and Stability

To enhance model generalization and prevent overfitting, especially given the relative depth of the network, **Dropout** layers are strategically placed within the `ConvBlock` structure. The dropout rates were progressively increased towards the bottleneck, where feature maps are most condensed, peaking at 0.3:

$$\text{Encoder Dropout Rates: } 0.1 \to 0.1 \to 0.2 \to 0.2 \to 0.3$$

Furthermore, the model employs **He/Kaiming Initialization** for its convolutional and transposed convolutional layers, aiding in stable training by ensuring signals propagate correctly through the deep network.

2.2.3 Feature Extraction for Bonus Task

A dedicated method, `extract_features`, is included in the **CustomUNet** class. This function returns the feature map from the **bottleneck layer**, which is a highly compressed representation of the input image. This feature map is specifically designed to be reused for the classification sub-task outlined in **Bonus A: detecting a specific character**.

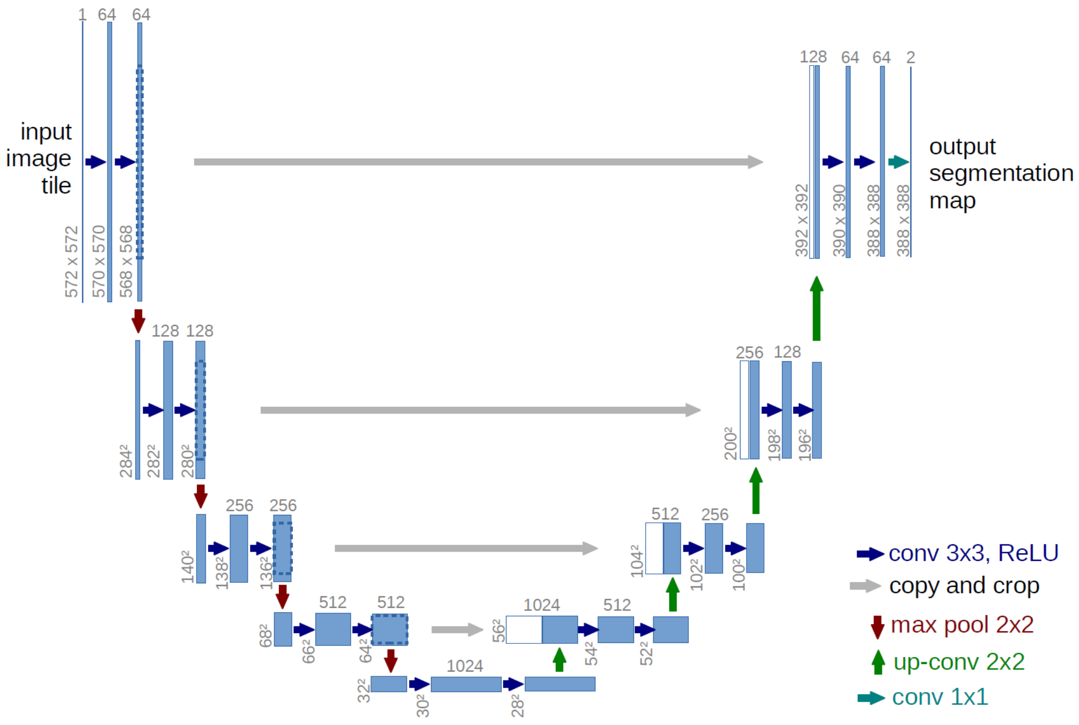

Figure 6: U-NET architecture

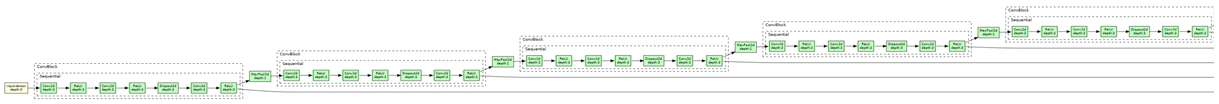

Figure 7: U-Net Custom Architecture: Encoder

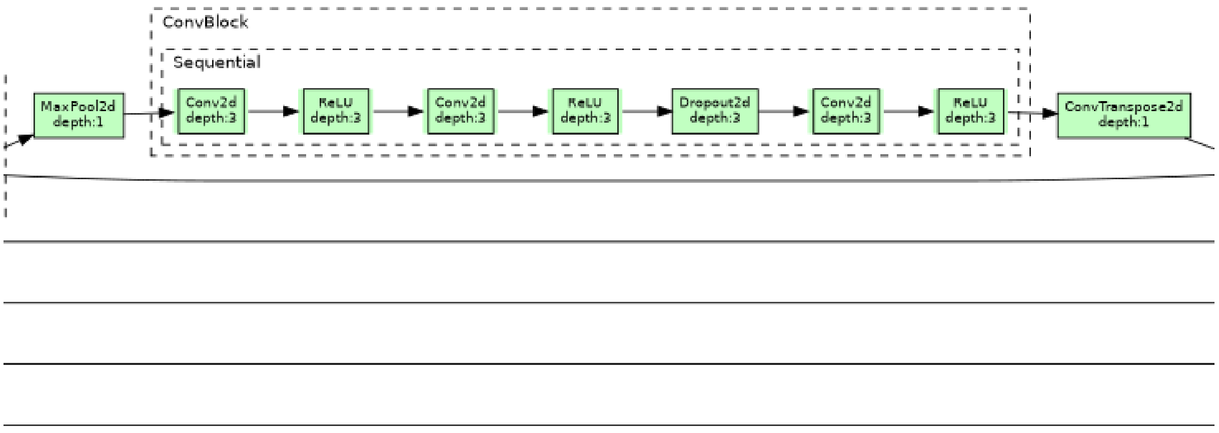

Figure 8: U-Net Custom Architecture: Bottleneck

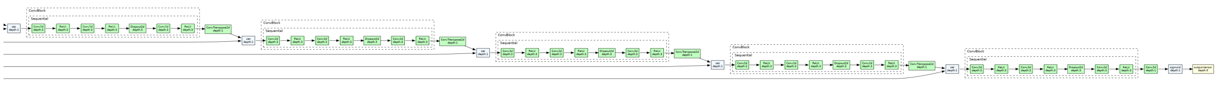

Figure 9: U-Net Custom Architecture: Decoder

3 Testing and Evaluation

This section details the training strategy, hyperparameter tuning, and performance evaluation methods used to train both segmentation models.

3.1 Data Management and Pre-processing

3.1.1 Dataset Implementation and Dynamic Resizing

Initial experiments with the raw image resolution (2048 × 2048) proved unfeasible due to high GPU memory consumption (as indicated by the need to reduce the batch size in previous attempts). The data loading pipeline is managed by the custom `ImageMaskDataset` class. This implementation is crucial as it integrates lazy loading with a pre-processing sequence designed to handle the high resolution of the source images and the complexity of the bounding box annotations:

1. **Annotation Rasterization at Original Resolution**: All polygon annotations (both Chinese characters and "ignore" regions) are first rasterized into pixel masks at the **original image resolution**. This high-fidelity rasterization ensures that even the smallest character bounding boxes are accurately represented before any downsampling occurs.
2. **Mask Filtering**: The final ground truth mask is generated by ensuring that a pixel belongs to a Chinese character bounding box **AND** does not overlap with any explicitly marked "ignore" regions.
3. **Dynamic Resizing to 512 × 512**: To mitigate Out-of-Memory errors and stabilize the training process, the images are resized from their original high resolution down to the designated **TARGET_IMG_HEIGHT** = 512 and **TARGET_IMG_WIDTH** = 512.

   ◇ **Image Resizing**: `cv2.INTER_AREA` interpolation is used for the image data (RGB), as it provides the most effective anti-aliasing and highest quality downsampling.

   ◇ **Mask Resizing**: `cv2.INTER_NEAREST` interpolation is strictly used for the binary mask data. This choice is deliberate, as it prevents the introduction of non-binary (fractional) pixel values, which could blur the class boundaries or dilute the signal from small character regions.
4. **Tensor Conversion**: The resized image is normalized to $[0, 1]$ and transposed to the PyTorch standard format (C × H × W), while the mask is converted to a single-channel float32 tensor (1 × H × W), ready for model input.

3.1.2 Dataset Split

The official training data was split into training and validation sets for monitoring generalization performance.

◇ **Training Set**: 845 images were used for model training.

◇ **Validation Set**: 155 images were reserved for validation and monitoring during training.

3.2 Hyperparameters and Optimization

The following hyperparameters were selected and used for training both model architectures:

- **Batch Size (BATCH_SIZE)**: **8** (Chosen to balance memory usage at 512×512 resolution with stable gradient estimation).
- **Number of Epochs (NUM_EPOCHS)**: **100** (A standard, generous starting value to ensure convergence).
- **Learning Rate (LEARNING_RATE)**: $\mathbf{1 \times 10^{-4}}$ (A typical starting value for the **Adam** optimizer, known for stable convergence).
- **Optimizer**: The **Adam** optimization algorithm was employed.

3.3 Addressing Class Imbalance: The Loss Function

The main challenge in this segmentation task is the severe **class imbalance**. Pixels belonging to the character bounding boxes are vastly outnumbered by background pixels. Training with standard **Binary Cross-Entropy (BCE)** loss heavily penalizes the incorrect classification of the background, leading to models that predict mostly zero (IoU close to zero).

3.3.1 Implementation of Dice Loss

To prioritize the correct segmentation of the small target regions, the custom **Dice Loss** was adopted.
The Dice Loss is derived from the **Dice Coefficient** (or F1 Score) and is highly sensitive to the overlap between the prediction (P) and the ground truth (G):

$$\text{Dice Coefficient} = \frac{2 \cdot |\mathrm{P} \cap \mathrm{G}|}{|\mathrm{P}| + |\mathrm{G}|}$$

The loss function is defined as: $\text{Loss} = 1 - \text{Dice Coefficient}$. Minimizing this loss is equivalent to maximizing the spatial overlap.

3.3.2 Sigmoid Activation

It is crucial to note that, unlike PyTorch's built-in **BCEWithLogitsLoss**, the custom **DiceLoss** implementation must explicitly apply the **Sigmoid function** to the raw network outputs (logits) before calculating the coefficient, ensuring the input values are normalized probabilities ($\in [0, 1]$).

3.4 Evaluation Metric

The primary evaluation metric used to assess the models' performance is the **Intersection over Union (IoU)** coefficient, also known as the Jaccard index. This metric directly measures the quality of the spatial overlap between the predicted mask and the ground truth mask. It is defined as:

$$\text{IoU} = \frac{|\mathrm{P} \cap \mathrm{G}|}{|\mathrm{P} \cup \mathrm{G}|}$$

A high IoU value indicates a strong performance in object localization. Preliminary results using only BCE and accuracy yielded an extremely poor IoU of 0.0029, confirming the necessity of the Dice Loss and IoU as the primary metric.

3.5 Qualitative Analysis and Visual Inspection

To gain insight into the models' behavior and the quality of their boundary predictions, a qualitative analysis was performed by visualizing the output of both **SimpleSegNet** and **CustomUNet** on a randomly selected image from the validation set.

3.5.1 Visualization Procedure

The procedure for visualization involves:

1. **Prediction**: Running a forward pass on a single image batch in evaluation mode (`unet_model.eval()`) to obtain the raw probability map (pred_tensor).
2. **Visualization**: Plotting the original image, the ground truth mask, the model's soft prediction (probabilities $\in [0, 1]$), and finally, an overlay of the soft prediction mask on the original image.

3.5.2 SimpleSegNet Visualization

For **SimpleSegNet**, the predictions generally show that the model successfully identifies the areas containing text. However, due to its structure which relies only on pooling indices for boundary recovery, the predicted boundaries tend to be less precise, often showing a "blob-like" quality, particularly around small or thin characters.

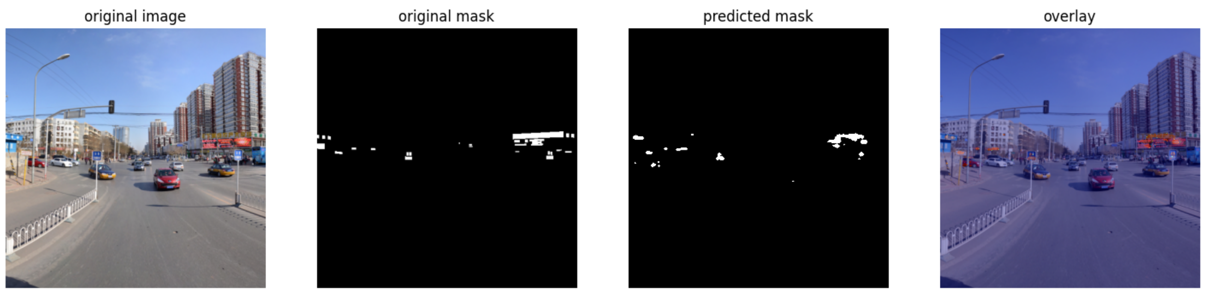

Figure 10: Qualitative results for **SimpleSegNet** on a validation image, showing the Original Image, Ground Truth, Soft Prediction Mask, and the Overlay. Note the smoother boundaries compared to U-Net.

3.5.3 CustomUNet Visualization

The predictions from the **CustomUNet** demonstrate better fidelity and closer alignment to the ground truth. The U-Net architecture's use of **skip connections** (**concatenation**) allows it to integrate high-resolution features from the early encoder stages with deep semantic features from the bottleneck. This results in:

- **Higher Recall**: The model is generally better at detecting all instances of the character bounding boxes.
- **Sharper Boundaries**: The predicted contours in the overlay are typically sharper and more accurate, which directly contributes to its higher IoU score (≈ 0.346).

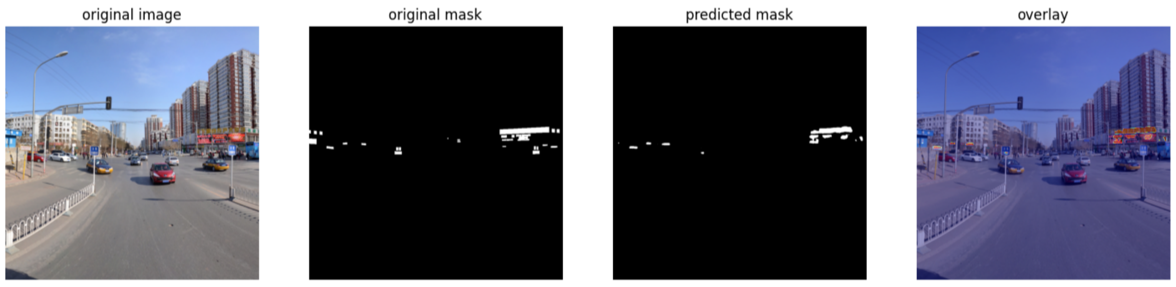

Figure 11: Qualitative results for **CustomUNet**. The predicted mask and the final overlay show sharper and more precise localization compared to SimpleSegNet.

4 Training Progress and Results

The two proposed models, **SimpleSegNet** and **CustomUNet**, were trained for 100 epochs using the defined hyperparameters: BATCH_SIZE = 8, LEARNING_RATE = 1×10^{-4}, and the **Dice Loss** function.

4.0.1 Quantitative Comparison of Model Performance

The final validation performance demonstrates that the **CustomUNet** architecture achieved superior results in localizing the character bounding boxes, which is attributed to its deeper structure and the effectiveness of skip connections in combining fine-grained and contextual features.

Table 1: Summary of Final Validation Results

Model	Final Val IoU	Final Val Dice Loss	Avg Time per Epoch (s)
SimpleSegNet	**0.3025** (Best: Epoch 96)	0.5439 (Epoch 96)	$\approx$ 16 s
CustomUNet	**0.3729** (Best: Epoch 95)	0.4873 (Epoch 95)	$\approx$ 48 s

1. **Effectiveness of Dice Loss**: Both models exhibited a major performance increase compared to the initial attempts with Binary Cross-Entropy (BCE) Loss. This confirms that the switch to **Dice Loss** successfully mitigated the effects of the class imbalance problem.
2. **Architecture Comparison**: The **CustomUNet** consistently outperformed **SimpleSegNet** on the validation set, achieving a peak IoU of **0.3729** versus SimpleSegNet's **0.3025**.
3. **Trade-off**: While **CustomUNet** is more accurate, it requires significantly more computational resources, taking approximately three times longer per epoch ($\approx$ 48 seconds) than **SimpleSegNet** ($\approx$ 16 seconds). This difference is expected, given U-Net's deeper structure.

4.0.2 Analysis of Convergence

The training logs show that both models achieved their peak performance before the 100th epoch:

◇ **SimpleSegNet**: Best val_iou reached at **Epoch 96** (**0.3025**), with subsequent epochs showing minor fluctuations. The gap between train_loss ($\approx$ 0.15) and val_loss ($\approx$ 0.55) is wide, suggesting that the model is **underfitting** or that the small target regions make the loss calculation very volatile on the validation set.

◇ **CustomUNet**: Best val_iou reached at **Epoch 95** (**0.3729**). The higher train_loss compared to SimpleSegNet, along with the lower final val_loss ($\approx$ 0.49), suggests a more stable convergence and better generalization.

5 Conclusion

The project successfully addressed the pixel-wise bounding box detection task using two distinct deep learning architectures. The challenge of severe class imbalance was effectively mitigated by switching the optimization criterion to the **Dice Loss**. Among the two models, the **CustomUNet** provided the best performance by leveraging its multi-scale skip connections, at the cost of significantly higher computational complexity. Future work could involve hyperparameter tuning for further performance gains or exploring cascaded detection methods.

6 Bonus Activity A: Transfer Learning for Character Classification

This bonus activity demonstrates the principle of **Transfer Learning** by repurposing the trained semantic segmentation model (**CustomUNet**) as a feature extractor for a binary classification task.

6.1 Task Definition

The goal is to classify an entire input image based on the presence or absence of the **most frequent Chinese character** found in the training dataset.

1. **Character Identification**: An initial data analysis was performed using `find_most_common_character` on the training annotations. This determined the target character and confirmed the existence of a severe class imbalance in the classification labels as well.
2. **Classification Labels**: For any given image, the label is set to 1 (Positive) if the most frequent character is present, and 0 (Negative) otherwise.

6.2 Model Architecture: FeatureClassifier

The dedicated classification model, `FeatureClassifier`, utilizes the knowledge learned by the segmentation network:

- **Backbone (CustomUNet)**: The encoder part of the pre-trained **CustomUNet** is used to extract high-level, dense feature maps. Crucially, **all parameters of the U-Net backbone are frozen** (`param.requires_grad = False`). This prevents the heavy segmentation weights from being corrupted by the new, lighter classification task and significantly speeds up training.
- **Feature Map**: The features are extracted from the U-Net's **Bottleneck layer**.
- **Classification Head**: A lightweight head is attached to this backbone:
  - **Global Average Pooling**: `nn.AdaptiveAvgPool2d(1)` reduces the 512×spatial_size feature map to a fixed-size vector of **512** features.
  - **Dense Layers**: This is followed by a **512 → 128** linear layer, a **128 → 1** output linear layer (producing logits), and a Dropout(**0.5**) layer for regularization.

6.3 Training and Evaluation Strategy

Due to the persistent class imbalance (e.g., $N_{\text{positives}}$ vs. $N_{\text{negatives}}$), the training required specialized techniques:

1. **Loss Function**: **nn.BCEWithLogitsLoss** was chosen, as it is numerically stable and allows the direct use of **pos_weight** for class weighting.
2. **Class Weighting**: The weight for positive samples was computed as the ratio of negatives to positives (pos_weight $= N_{\text{negatives}}/N_{\text{positives}}$). This dynamically balances the contribution of the minority class to the total loss, ensuring the model does not ignore the positive samples.
3. **Metrics**: The F1 Score was used as the primary metric for evaluation and checkpointing, as it provides a robust measure for imbalanced classification tasks (unlike simple Accuracy).

6.4 Results Summary

The classification head was then trained for 100 epochs with weighted loss.

- ◇ **Best Val F1 Score**: **0.4255** (Epoch 97)
- ◇ **Final Val Accuracy**: 0.6516

This transfer learning experiment validates that the features learned by the **CustomUNet** encoder for pixel-wise segmentation are relevant and transferable to the image-level classification task, achieving good performance with minimal training time due to the frozen backbone.

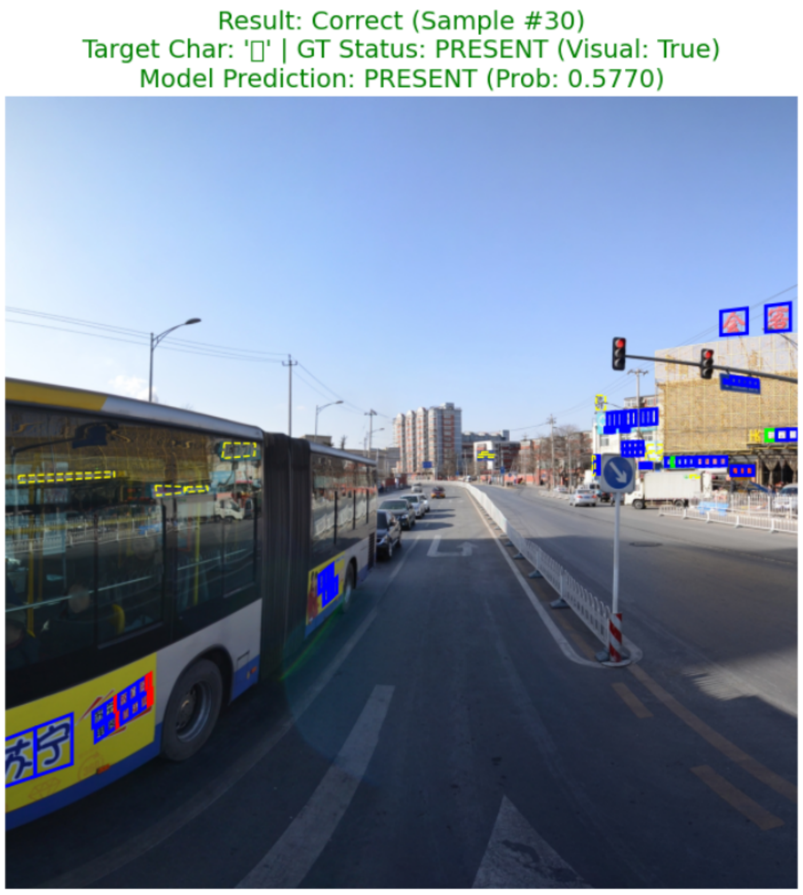
Result: Correct (Sample #30)
Target Char: '□' | GT Status: PRESENT (Visual: True)
Model Prediction: PRESENT (Prob: 0.5770)

Figure 12: Classification Example for **CustomUNet Feature Classifier**. The predicted label tells that the character (中) is present and correspond to the ground truth. The character is represented by the green boxes

7 Code Execution and Project Structure

The entire project pipeline, from data loading to model training and evaluation, is implemented in Python using the PyTorch framework. The code is structured across several distinct Jupyter Notebooks.

7.1 Execution Environment and Data Access

- ◇ **Required Environment**: All code **must** be executed directly on the **mltgpu-2 server**.
- ◇ **Dataset Source**: The data loading functions are configured to automatically retrieve the image files and annotation metadata from the required path (`/srv/data/lt2326-h25/a1/images` and the corresponding `jsonl` files).

7.2 Project Notebooks Overview

The training and analysis tasks are divided into three separate Jupyter Notebooks:

1. `SegNet.ipynb`: Contains the implementation and training loop for the `SimpleSegNet` architecture.
2. `UNet.ipynb`: Contains the implementation and training loop for the `CustomUNet` architecture, including the setup for the **Bonus A: FeatureClassifier**.
3. `Visualize_results.ipynb`: Used for the qualitative analysis, allowing the user to load the trained models and visualize the predicted masks overlaid on validation images.

7.3 Usage Instructions

To ensure proper execution and reproducibility:

- ◇ **Sequential Execution**: In each training notebook, all code cells must be executed **sequentially, from top to bottom**. The dataset loading, model definition, training loop, and saving of checkpoints are dependent on the previous cells.
- ◇ **Code Documentation**: Every crucial block of code, including model definitions, dataset resizing logic, loss function setup (e.g., Dice Loss), and training loops, is accompanied by **detailed comments** explaining its function.