Road_Segmentation

June 1, 2025

Semantic Segmentation Competition (30%)

For this competition, we will use a small autonomous driving dataset. The dataset contains 150 training images and 50 testing images.

We provide baseline code that includes the following features:

- Loading the dataset using PyTorch.
- Defining a simple convolutional neural network for semantic segmentation.
- How to use existing loss function for the model learning.
- Train the network on the training data.
- Test the trained network on the testing data.

The following changes could be considered:

1. Data augmentation
2. Change of advanced training parameters: Learning Rate, Optimizer, Batch-size, and Dropout.
3. Architectural changes: Batch Normalization, Residual layers, etc.
4. Use of a new loss function.

Your code should be modified from the provided baseline. A pdf report of a maximum of two pages is required to explain the changes you made from the baseline, why you chose those changes, and the improvements they achieved.

Marking Rules:

We will mark the competition based on the final test accuracy on testing images and your report.

Final mark (out of 50) = accuracy mark + efficiency mark + report mark

Accuracy Mark 10:

We will rank all the submission results based on their test accuracy. Zero improvement over the baseline yields 0 marks. Maximum improvement over the baseline will yield 10 marks. There will be a sliding scale applied in between.

Efficiency Mark 10:

Efficiency considers not only the accuracy, but the computational cost of running the model (flops: https://en.wikipedia.org/wiki/FLOPS). Efficiency for our purposes is defined to be the ratio of accuracy (in %) to Gflops. Please report the computational cost for your final model and include the efficiency calculation in your report. Maximum improvement over the baseline will yield 10 marks. Zero improvement over the baseline yields zero marks, with a sliding scale in between.

Report mark 30:

Your report should comprise:

1. An introduction showing your understanding of the task and of the baseline model: [10 marks]
2. A description of how you have modified aspects of the system to improve performance. [10 marks]

   A recommended way to present a summary of this is via an "ablation study" table, eg:

Method1	Method2	Method3	Accuracy
N	N	N	60%
Y	N	N	65%
Y	Y	N	77%
Y	Y	Y	82%

3. Explanation of the methods for reducing the computational cost and/or improve the trade-off between accuracy and cost: [5 marks]
4. Limitations/Conclusions: [5 marks]

1 Download Data & Set Configurations

1.1 Download the Dataset

Dowanload & Unzip the Dataset.

1.2 Set Configurations

1.3 Edit Configurations to Tune Model Performance

This section allows you to modify core training parameters such as the learning rate, batch size, image dimensions, and number of training epochs. These hyperparameters can significantly affect model performance and should be tuned based on your dataset and hardware constraints. The code also includes basic checks to ensure that the data path is valid and that CUDA is available for GPU acceleration.

2 Define a Dataloader to Load Data

This class handles the loading of images and their corresponding labels while applying systematic data transformations. Since the dataset is small, effective data augmentation such as flipping, resizing, rotation, brightness adjustments, and noise addition is used to improve model generalisation. The transformations ensure that both images and masks are consistently processed and converted into PyTorch tensors for training. The loader also includes sanity checks to confirm data integrity and correct alignment between images and labels.

3 Define a Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for processing data with a grid-like topology, such as images. They leverage convolutional layers to automatically and adaptively learn spatial hierarchies of features through backpropagation. Key components include:

- **Convolutional Layers:** Apply learnable filters/kernels that slide across the input to produce feature maps, capturing spatial features like edges, textures, and shapes.
- **Pooling Layers:** Downsample feature maps to reduce spatial dimensions and computation, while providing translation invariance.
- **Activation Functions:** Non-linear functions *(e.g. ReLU)* applied after convolutions to introduce non-linearity, enabling the model to learn complex patterns.
- **Fully Connected Layers:** Typically at the network's end for classification tasks, mapping extracted features to output labels.
- **Advantages:** CNNs reduce the number of parameters compared to fully connected networks by weight sharing and local connectivity, making them highly effective for image-related tasks.

3.1 Building a New CNN Model

BetterSegNetwork is a deep convolutional encoder-decoder architecture tailored for semantic segmentation, inspired by U-Net designs with residual connections and dilated convolutions. The model's structure and design choices include:

- **Encoder:** Extracts hierarchical features with increasing channel depth. Uses residual blocks and dilated convolutions to expand receptive fields without losing resolution.
- **Center Block:** Enhances contextual feature learning via increased dilation, capturing broader image context.
- **Decoder:** Gradually upsamples and fuses encoder features through skip connections to reconstruct fine spatial details for pixel-level segmentation.
- **Regularisation:** Batch normalisation and dropout are used throughout to improve training stability and reduce overfitting.
- **Transfer Learning:** Pretrained weights are employed in the encoder to leverage learned features from large datasets, accelerating convergence and improving generalisation on limited data.
- **Output:** Final layers generate per-pixel class probabilities for accurate segmentation maps.

This combination of design elements balances model complexity with generalization, making BetterSegNetwork suitable for challenging segmentation tasks, especially when training data is scarce.

3.2 Making a Pre-Trained Hybrid Model

This model architecture combines EfficientNet-B3 as the encoder backbone with SegFormer as the decoder for semantic segmentation:

- **Encoder Backbone:** EfficientNet-B3 pretrained on ImageNet, providing efficient and robust feature extraction.
- **Decoder:** SegFormer decoder, known for its efficient and accurate multi-scale feature aggregation without requiring positional encoding.
- **Device Configuration:** Automatically selects GPU if available, otherwise runs on CPU.
- **Model Initialisation:** Uses `segmentation_models_pytorch` (SMP) library's SegFormer implementation with EfficientNet-B3 encoder and 19 output segmentation classes.
- **Pretrained Weights Loading:** Loads pretrained weights from a checkpoint if available, while handling possible 'module.' prefixes from distributed training.
- **Model Summary:** Outputs a detailed summary of the model architecture for the specified input size.

This hybrid approach leverages EfficientNet-B3's strong feature extraction and SegFormer's efficient transformer-based decoding, enabling high-quality segmentation with reduced computational cost.

3.3 Loss & Optimisation

3.3.1 Weighted Cross Entropy Loss

Weighted Cross Entropy (WCE) is a modification of the standard Cross Entropy loss that introduces class-specific weighting factors to address the issue of class imbalance, which is common in real-world segmentation datasets. In typical urban scene datasets (e.g., Cityscapes-like), certain classes such as *"road"* or *"building"* may dominate the pixel distribution, while others like *"traffic light"* or *"person"* may be underrepresented. This imbalance can lead the model to bias its predictions toward the majority classes.

By assigning higher weights to minority classes, WCE increases the penalty for misclassifying them, thereby encouraging the model to learn more balanced representations. The weights are typically computed as the inverse frequency of each class or using a smoothed variant such as median frequency balancing.

In this implementation, the WCE loss was applied over the softmax outputs of the model, with class weights provided as a fixed tensor based on label statistics from the training dataset. This approach maintains the interpretability and differentiability of standard Cross Entropy while improving performance on underrepresented classes.

3.3.2 Multi-Class Dice Loss

To address the challenges of class imbalance and to better reflect the spatial structure of predicted segments, a custom Multi-Class Dice Loss function was implemented. Unlike standard Cross Entropy, which penalises misclassifications on a per-pixel basis, Dice Loss directly optimises for region-level similarity between the predicted and ground truth masks. This is particularly useful in semantic segmentation tasks where smaller or less frequent classes may be overwhelmed by dominant ones.

The implemented loss extends the standard Dice formulation to multi-class settings by calculating the Dice coefficient (or F-score) per class after applying a softmax over class logits. Ground truth labels are one-hot encoded and aligned with the prediction tensor, while ignore labels (255) are excluded from the calculation. A smoothing term ( ) is included for numerical stability, and the formula is parameterised by a ^{2} term to allow future adjustment of precision-recall trade-offs. In this instance, was set to 1.0, placing equal emphasis on both precision and recall. The final loss is computed as the sum of Dice errors across all valid classes.

3.3.3 Dice Loss With Cross Entropy

This combined loss function was used to balance pixel-wise classification accuracy with region-level overlap. Cross Entropy Loss focuses on penalising incorrect class predictions at the pixel level, while Dice Loss directly optimises for the similarity between the predicted and ground truth regions. By combining the two with equal weighting (0.5 each), the model was encouraged to perform well both at the granular pixel scale and in capturing the overall structure of objects. This approach was particularly beneficial in handling class imbalance and improving segmentation performance on smaller or less frequent classes, which may be underrepresented in the dataset.

3.3.4 Implementation of Chosen Loss Function & Optimiser

The segmentation model was trained using a combination of a selected loss function and an appropriate optimiser to ensure stable convergence and effective learning. The loss function used was [INSERT LOSS FUNCTION NAME HERE], chosen for its ability to [INSERT REASON — e.g., address class imbalance, improve region overlap, etc.]. For optimisation, the AdamW optimiser was applied, with a learning rate of [INSERT LEARNING RATE] and weight decay to promote generalisation. The choice of optimiser and hyperparameters was guided by the need to balance convergence speed with stable updates over the course of training.

4 The Function Used to Compare the Precision

DO NOT MODIFY THIS CODE!

5 Define Functions to Get & Save Predictions

This section contains utility functions to manage prediction outputs during model evaluation. It includes creating necessary folders, moving temporary prediction files to final result directories, and converting grayscale masks into colorised images for better visualisation. The main function runs the model on test images, saves both grayscale and color predictions, and prepares them for further analysis or display.

6 Train the Network

6.1 Training & Testing Functions

Utility functions for handling model predictions and output management during evaluation. These functions create necessary directories, move prediction files from temporary folders to final result locations, and convert grayscale segmentation masks into colorised images using a predefined palette for easier visualisation. The main evaluation function runs inference on test images, saves both grayscale and colorised predictions, and prepares the outputs for accuracy calculations or further analysis.

6.2 Training Loss & IoU Performance

The first plot shows the training loss over iterations, illustrating how the model's error decreases during training. The second plot tracks the Intersection over Union (IoU) metric from epoch 101 onward, reflecting improvements in segmentation accuracy as training progresses. Together, these visualisations help assess model learning and performance over time.

6.3 Randomised Prediction Test Results from the Model:

This function randomly selects a few predicted segmentation masks and their corresponding original images, then displays them side-by-side. It helps qualitatively assess the model's performance by providing a clear comparison between the input images and the predicted outputs.

7 FLOPs

This section calculates the total number of floating point operations (FLOPs) the model performs during a single forward pass, providing a measure of its computational cost. Using the fvcore library, we quantify the model's complexity in GFLOPs, which helps compare efficiency across different architectures.

In deep learning, FLOPs *(Floating Point Operations)* quantify the total number of arithmetic operations—such as additions, multiplications, and divisions—that a model performs during a single forward pass *(i.e. when making a prediction)*. This metric serves as an indicator of a model's computational complexity. When discussing large-scale models, FLOPs are often expressed in GFLOPs *(Giga Floating Point Operations)*, where 1 GFLOP equals one billion operations. This unit helps in comparing the computational demands of different models.

DO NOT MODIFY THIS CODE!