Deep Learning for Visual Computing

Assignment 2

Dzhamilia Kulikieva (12340251)
Nooreldin Lasheen (12302427)
Group 3

June 2025

Contents

1 Introduction

In this assignment, we explored semantic segmentation, a task that assigns a class label to each pixel in an image. Using the OxfordIIITPet and Cityscapes datasets, we trained and evaluated several models, including PyTorch's pre-trained FCN-ResNet50 and the transformer-based SegFormer. We also investigated the impact of pre-training and fine-tuning, comparing different strategies to improve segmentation performance.

2 Part 4: Using PyTorch's Pre-trained FCN

2.1 Overview

In this part, we trained the fcn_resnet50 model on the OxfordIIITPet dataset - first from scratch, and then using pre-trained weights for the encoder. We monitored training and validation loss as well as mIoU to compare both approaches.

2.2 Training Setup

Parameter	Value
Dataset	OxfordIIITPet
Model	FCN-ResNet50
Loss Function	CrossEntropyLoss (ignore_index=255)
Optimizer	AdamW (lr = 0.001, amsgrad=True)
Learning Rate Scheduler	ExponentialLR (gamma = 0.98)
Epochs	30
Batch Size	64
Validation Frequency	Every 2 epochs

Table 1: Training setup for FCN-ResNet50 on OxfordIIITPet dataset

2.3 Results: FCN-ResNet50 on OxfordIIITPet

Figure 1: Train/Val Loss and mIoU for FCN

The training and validation performance was evaluated using loss and mean Intersection-over-Union (mIoU) metrics.

Table 2: Comparison of FCN training with and without pre-trained weights (on Oxford-IIIT Pet)

Model	Train mIoU	Val mIoU	Train Loss	Val Loss
FCN from Scratch	0.7951	0.5874	0.1532	0.7797
FCN (Pretrained)	**0.8337**	**0.6995**	**0.1170**	**0.4638**

The best validation mIoU (0.6995) for the pre-trained model was achieved at epoch 27.

2.3.1 FCN Performance Overview

The FCN model using a pre-trained backbone consistently performs better than the version trained from scratch across all metrics, as shown in Table 2. Better generalization is demonstrated by the validation mIoU improving by almost 11%. Furthermore, the pretrained model achieves lower training and validation loss values and converges more quickly. This demonstrates how successful transfer learning is, particularly when dealing with small datasets like the Oxford-IIIT Pet dataset.

The scratch model has a wider divergence between training and validation measures, suggesting weaker generalization, even though both models achieve good training performance. The pre-trained model, on the other hand, maintains better consistency between the training and validation results.One limitation is the downscaling of the photos to 64×64, which is could compromise the accuracy of the fine-grained segmentation.

3 Part 5: SegFormer Implementation

3.1 Overview

SegFormer is an efficient encoder-decoder architecture for semantic segmentation. Its encoder, based on the MixVisionTransformer (MiT), extracts multi-scale features using overlapping patch embeddings and spatially reduced attention. The decoder (SegFormerHead) projects these features into a common space, aligns their resolutions, and fuses them to generate the final segmentation map. The main components include the `Block` (attention and MLP with depthwise convolution), `OverlapPatchEmbed` (for patch tokenization), and `SegFormerHead` (for feature fusion and prediction).

3.2 Implemented Parts

Table 3: Manually Implemented Components of the SegFormer Model

Module	Implementation Description
`Block` (`mit_transformer.py`)	Completed the `forward()` function to apply layer normalization, spatial-reduction attention, and an MLP block with depthwise convolution. Residual connections were added after both attention and MLP sub-blocks.
`OverlapPatchEmbed` (`mit_transformer.py`)	Implemented the projection of image patches into overlapping tokens using a strided convolution. The flattened tokens were then passed through a `LayerNorm` to produce the embedded sequence.
`SegFormerHead` (`segformer_head.py`)	In the decoder head, completed the forward function to transform and align multi-scale encoder outputs. Features were linearly embedded, interpolated to a common resolution, concatenated, and fused via a 1×1 convolution followed by batch normalization and ReLU activation.

3.3 Training SegFormer from Scratch

Same setup as in Part 4

Challenges and Assumptions It was necessary to carefully upsample feature maps from several encoder stages to a common resolution in order to fuse them. Close attention was needed to manage permutations and tensor reshaping across MLP layers. A fixed 64×64 input resolution was also specified in order to conserve memory, which made implementation easier but reduced the accuracy of fine-grained segmentation.

3.4 Results: SegFormer on OxfordIIITPet

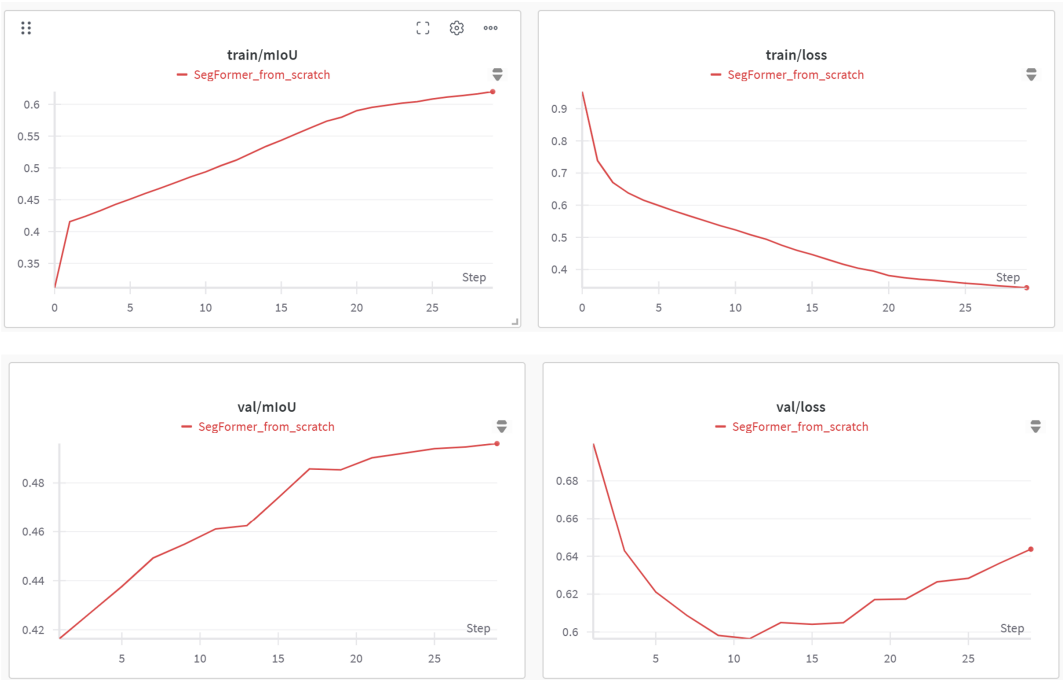

Figure 2: Train/Val Loss and mIoU for FCN

Table 4: Performance of SegFormer trained from scratch on the segmentation task

Model	Train mIoU	Val mIoU	Train Loss	Val Loss
SegFormer (from Scratch)	**0.6199**	**0.4961**	**0.3432**	**0.6437**

3.4.1 SegFormer Performance Overview

The SegFormer model, trained from scratch on the Oxford-IIIT Pet dataset, reached a final validation mIoU of 0.4961 and a training mIoU of 0.6199. Throughout training, both losses steadily decreased, with final training and validation losses of 0.3432 and 0.6437, respectively, as illustrated in Figure 2.These results reflect stable convergence and reasonable generalization, despite the absence of pre-training.

Although SegFormer underperforms compared to the pretrained FCN in terms of validation mIoU, it benefits from a more modern transformer-based design and lightweight decoder. Its use of overlapping patch embeddings and spatial-reduction attention enables effective multi-scale feature extraction. However, the reduced input resolution (64×64) likely limited its ability to capture fine boundary details, which is a notable constraint for fine-grained segmentation tasks.

4 Part 6: Pre-train on Cityscapes, Fine-tune on OxfordPets

4.1 Overview

In this part, we pre-trained the SegFormer model on the Cityscapes dataset and then fine-tuned it on OxfordIIITPet to assess the benefits of transfer learning. Since the datasets have different label sets, only the encoder weights were reused, while the decoder was trained from scratch. We compared two fine-tuning strategies - with and without freezing the encoder - and evaluated their impact on segmentation performance.

Additionally, we explored the effect of different learning rates during fine-tuning, comparing the results with `lr = 0.001` and `lr = 0.0005` while keeping the encoder frozen.

4.2 Pre-training on Cityscapes

We pre-trained the SegFormer encoder on the Cityscapes dataset for 31 epochs, excluding pixels with label 255 from the loss. The learned encoder weights were saved for later use in fine-tuning on the OxfordIIITPet dataset.

4.2.1 Pre-training Setup

Parameter	Value
Dataset	Cityscapes (19 classes)
Model	SegFormer
Loss Function	CrossEntropyLoss (ignore_index = 255)
Epochs	31
Batch Size	64

Table 5: Pre-training setup for SegFormer on Cityscapes

4.3 Fine-tuning on Oxford-IIIT Pet

Option A: Using encoder weights as initialization.
In this setting, we initialized the encoder of the SegFormer model with pre-trained weights from Cityscapes by using `model.net.encoder.load_state_dict(...)`. Both the encoder and decoder were then trained on the OxfordIIITPet dataset. This setup allows the encoder to adapt to the target domain while starting from a meaningful representation.

Option B: Freezing the encoder.
In this setup, we also loaded the pre-trained encoder weights from Cityscapes, but froze the encoder to prevent its parameters from being updated during training. This was done by setting `requires_grad = False` for all encoder parameters. Only the decoder was optimized, treating the encoder as a fixed feature extractor.

4.4 Results Comparison

Fine-tuning significantly improved segmentation performance compared to training the SegFormer model from scratch. The pre-trained model with a frozen encoder achieved a validation mIoU of 0.53, while fine-tuning the entire model reached 0.64. In contrast, training from scratch yielded only 0.17 mIoU.

Model	Train Loss	Train mIoU	Val Loss	Val mIoU
Pre-trained from scratch	0.4876	0.2581	0.9257	0.1695
Fine-tuned SegFormer	**0.2037**	**0.7558**	**0.4577**	**0.6439**
Frozen encoder (lr=0.001)	0.5081	0.5432	0.5344	0.5296
Frozen encoder (lr=0.0005)	0.5198	0.5343	0.5398	0.5245

Table 6: Comparison of training and validation performance for different fine-tuning strategies.

This highlights the importance of transfer learning, especially when training on smaller datasets. Additionally, freezing the encoder and fine-tuning only the decoder was less effective than updating the entire model, though it still outperformed the scratch baseline

The plots below additionally underline that fine-tuning SegFormer model (orange) outperforms both training from scratch (purple) and freezing the encoder (blue).

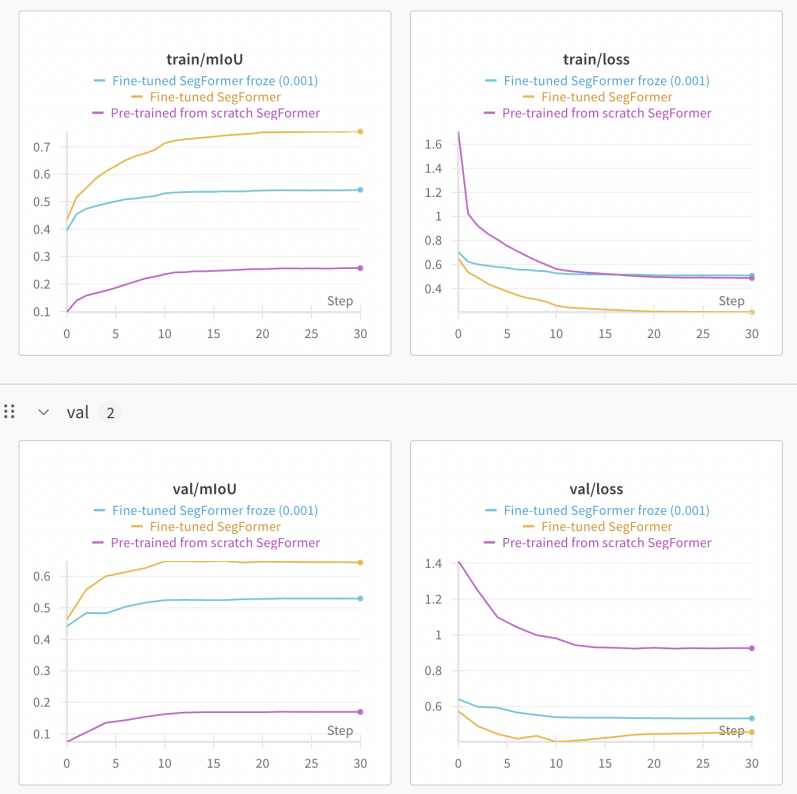

Figure 3: Comparison of Fine-tune Strategies with the SegFormer trained from scratch

4.4.1 Learning Rate Sensitivity Study

To get additional points, we compared two fine-tuning runs with a frozen encoder using different learning rates. Although the lower learning rate (0.0005) produced smoother training, the higher rate (0.001) yielded a better final result.

Learning Rate	val mIoU	Observation
0.001	52.96	Slightly better performance
0.0005	52.45	More stable, but lower final mIoU

Table 7: Comparison of learning rates for fine-tuning with frozen encoder

This suggests that while smaller learning rates are commonly recommended for fine-tuning, they may not always yield better results, especially when the number of training epochs is limited.

4.5 Predicted Masks

Figure 4: Sample input images from OxfordIIITPet validation set

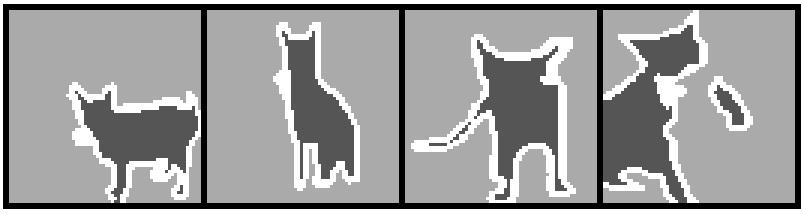

Figure 5: Ground truth segmentation masks

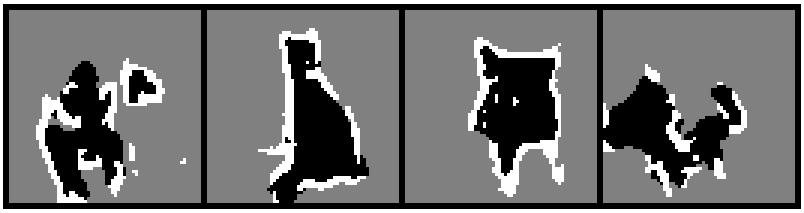

Figure 6: Predicted masks using fine-tuned SegFormer (best model)

The predicted masks show that the fine-tuned SegFormer model can accurately segment pet contours in most validation images. Despite the low resolution and limited training epochs, the model successfully distinguishes pets from the background, demonstrating the effectiveness of encoder pre-training.

5 Discussion

5.1 SegFormer vs. FCN

We experimented with the FCN (Fully Convolutional Network), a conventional architecture in addition to SegFormer. In contrast to SegFormer, which employs overlapping patch embeddings and transformer-based attention, FCN depends on convolutional backbones like as ResNet and upsampling via transposed convolutions. Better multi-scale representation and parameter efficiency are provided by SegFormer.

5.2 Use of Downsampling/Upsampling

In segmentation tasks, downsampling improves in reducing computational costs and expanding the receptive field—a crucial trade-off. Although downsampling is used by both FCN and SegFormer, SegFormer combines data from several encoder stages to restore spatial detail more effectively than FCN does by depending just on deconvolution procedures.

5.3 Pre-training and Fine-tuning

Pre-training helps by providing a strong initialization for model weights, enabling faster convergence and better generalization. It is especially useful for domain transfer or tasks with limited training data, where learning from scratch would underperform

5.4 Experimental Findings

Pre-training on Cityscapes significantly improved segmentation performance on OxfordIIITPet. Fine-tuning the entire model outperformed freezing the encoder. Lowering the learning rate led to smoother training but slightly worse final results compared to the default rate.

6 Bonus: Efficient Self-Attention

SegFormer improves standard self-attention by reducing the spatial size of keys and values using strided convolution when `sr_ratio > 1`. This reduces computation while retaining context, which is critical for high-resolution inputs. In mit_transformer.py, this is implemented as:

```
self.sr = nn.Conv2d(dim, dim, kernel_size=sr_ratio, stride=sr_ratio)
x_ = x.permute(0, 2, 1).reshape(B, C, H, W)
x_ = self.sr(x_).reshape(B, C, -1).permute(0, 2, 1)
x_ = self.norm(x_)
kv = self.kv(x_).reshape(B, -1, 2, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
```

7 Disclaimer

Sections 4 and 5 were completed by **Nooreldin Lasheen**, using his own implementations.Section 6 was independently reimplemented by **Dzhamilia Kulikieva**, including her own versions of the code for parts 4–6. The code used for Section 6 is included in this report, while the original implementation for Sections 4–5 is available via the GitHub link https://github.com/Nooryasser74/Assignment_2_deep_learning/tree/main

Important: If the attached code is executed, the results in Sections 4–5 may slightly differ from those originally reported.