CITY UNIVERSITY
LONDON

Deep Learning for Image Analysis

Written Report

Martin Fixman and Grigorios Vaitsas

2023/2024 Term

1 Introduction

In this study we have deployed the freely available Cityscapes dataset [1] to perform semantic segmentation analysis on urban scene imagery using a variety of Deep Learning architectures. Semantic segmentation is a computer vision technique used to identify and label specific parts of an image at pixel level. The main goal is to partition an image into areas (segments) that correspond to different objects or classes of interest (for example person, car, tree, road, building etc.). This technique is widely used in various fields like for instance, in autonomous driving, where it helps cars understand the environment around them, as well as in medical imaging, remote sensing, and robotics. Semantic segmentation is typically performed using deep learning methods, particularly convolutional neural networks (CNNs) and their variants as well as more recent attention-based models.

2 Dataset

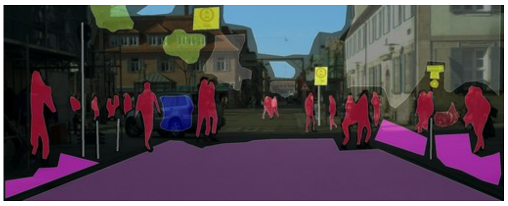
The Cityscapes dataset is a large-scale dataset widely used for training and evaluating algorithms in the fields of computer vision, particularly for semantic understanding of urban street scenes. It consists of high resolution images captured from a vehicle-mounted camera while driving through various cities in Germany, including Frankfurt, Munich and Strasbourg. The capturing apparatus that was used was equipped with a dual for stereoscopic capability, allowing for the possibility of applying stereo vision techniques useful for tasks like depth estimation, 3D reconstruction and scene understanding.

The images that are used in our analysis are located in the folder **leftImg8bit**. These are taken from the left camera of the stereo camera system. These are the main high-resolution images (2048 $\times$1024 pixels) with an 8-bit color depth that are used for all semantic and instance segmentation tasks. Each image is stored in a PNG file format with a size that varies between 2.0 and 2.5 MB. The images have an RGB color space, which means that there are 3 color channels for each image.

The dataset includes two different types of image annotations:
Fine annotations: These provide detailed, pixel-level annotations for 5,000 images which have labels for 30 classes such as road, car, pedestrian, building, and traffic light. This dataset is split into 2,975 annotations for training, 500 for validation and 1,525 for testing (see Figure 1(b)). It is worth mentioning that the ground truth annotations are not available to the users and this happens in order to allow for fair evaluation and benchmarking of models. Users instead need to submit their code to an online evaluation server which provides the quantitative performance metrics.

Coarse annotations: These include a much larger set of 20,000 images with less precise annotations intended for data augmentation and for tasks where granularity is not critical (see Figure 1(a) for an example). The coarse annotations are split into a training and validation set. No test set is provided since this is only done using the fine annotations set.

The scenes in our dataset represent a variety of urban settings, seasons, daylight


(a) Coarse mask

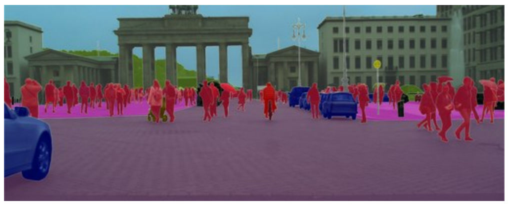

(b) Fine mask

Figure 1: Examples of coarse (a) and fine (b) mask annotations superimposed on the corresponding original images

conditions, and weather scenarios, providing robust, real-world environments for training models that need to perform under varied conditions. This dataset has been widely used in research for developing and testing algorithms on tasks such as object detection, semantic segmentation, and instance segmentation in urban settings as well as advancing the state-of-the-art in visual perception for autonomous driving systems. The Cityscapes dataset can be accessed in the following address:

`https://www.cityscapes-dataset.com`

Our particular approach in this study is to create and train at least two models; one with a basic architecture that will form our baseline and a second one where we will be exploring a more complex architecture. We are aiming to demonstrate first of all that both our models are able to perform semantic segmentation on the chosen dataset. Our second goal is to investigate how the choice of various hyper-parameters and tweaks in architecture can affect the accuracy and performance of the models.

3 Architectures

3.1 Fully Convolutional Netowrk

Modern, state-of-the-art techniques for image analysis involve convolutional neural networks (CNNs) in their implementation. CNNs can be used to extract vital information from spatial features, which allow us to classify and segment objects in images. A type of neural network architecture designed specifically for semantic segmentation is the Fully Convolutional Network (FCN). This architecture, developed by researchers from UC Berkeley in 2014 [4], has been influential in advancing the field of computer vision, particularly in tasks requiring dense prediction, like semantic segmentation.

Unlike standard convolutional neural networks used for image classification, which typically end with fully connected layers, FCNs are composed entirely of convolutional layers. This design allows them to take input of any size and output segmentation maps that correspond spatially to the input image, providing a per-pixel classification. FCNs transform the fully connected layers found in traditional CNNs (like those in AlexNet or VGGNet) into convolutional layers (Figure 2). This is done by treating the fully

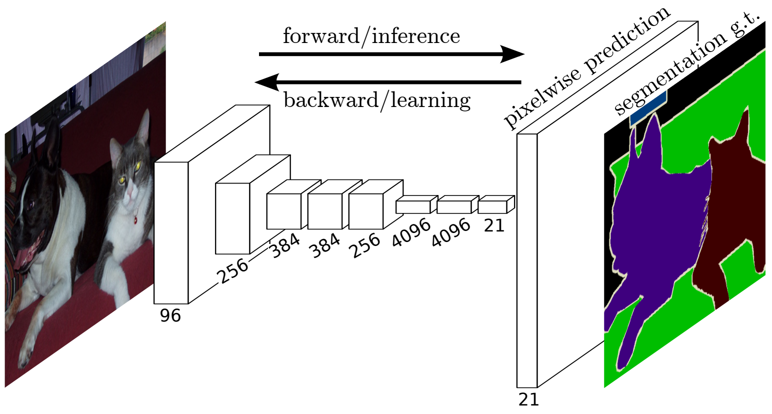


Figure 2: Example of a fully convolutional network used for semantic segmentation [4]

connected layers as convolutions with kernels that cover the entire input region. For example a fully connected layer that accepts an input of size 16×16 can be reimagined as a convolutional layer with a 16×16 filter size. To generate output segmentation maps that match the size of the input image, FCNs use transposed convolution layers (also known as deconvolutional layers) for upsampling. This process helps in recovering the spatial dimensions that are reduced during the pooling or convolutional operations in earlier layers. FCNs often utilize skip connections to combine deep, semantic information from lower layers with the shallow, appearance information in the upper layers of the network. This helps in improving the accuracy and detail of the output segmentation maps, as it allows the network to use both high-level and low-level features.

3.2 U-net

Further improving upon the architecture of the fully convolutional network, the U-net, first introduced in a paper by Olaf Ronneberger, Philipp Fischer and Thomas Brox in 2015 [5], has become very popular due to its efficiency and effectiveness, especially where data is limited. U-Net's architecture is distinctive because of its symmetric shape, which resembles the letter "U" (see Figure 3). It consists of two main parts: the left side of the U-net is the contracting path (encoder) which is designed to capture the context of the input image. This path is essentially a typical convolutional network that consists of repeated application of convolutions, followed by rectified linear unit (ReLU) activations, and max pooling for downsampling. Each of these steps helps in capturing the features at different levels of abstraction. On the right side of the network is the expansive path (Decoder), which performs the task of precise localization using transposed convolutions for upsampling. This part of the network gradually recovers the spatial dimensions and detail of the input image. The upsampling in the expansive path is combined with the high-resolution features from the contracting path via skip connections. These connections are crucial as they help in propagating context information to higher resolution layers, which enhances the accuracy of the output segmentation maps. U-Net's

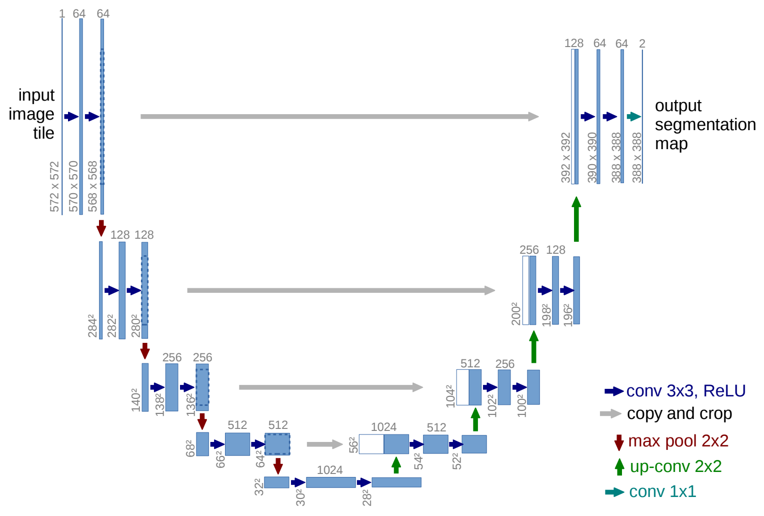


Figure 3: Example of the original U-net architecture as presented in [5]

design is particularly effective in providing good segmentation results even with small training datasets, which has led to its widespread adoption and adaptation in different segmentation tasks beyond its initial medical imaging focus.

3.3 Vision Transformer and Swin Transformer

The last architecture we explored in our study was a Vision Transformer (ViT), which is a novel approach to applying transformer models, primarily used in Natural Language Processing (NLP) tasks, to image recognition. In the paper "An Image is worth 16×16 Words: Transformers for Image Recognition at scale" [2], the authors proposed a method that treats an image as a sequence of fixed-size patches (similar to words in NLP). Each patch is embedded similarly to how tokens are embedded in NLP, and then processed by a standard transformer encoder. An overview of this model is depicted in Figure 4.

Vision transformers can achieve excellent results, outperforming many advanced CNN architectures, particularly when pre-trained on very large datasets. However, the computational complexity especially when applied on high-resolution images increases rapidly, especially for dense prediction tasks like semantic segmentation. For this reason, an improved approach has been suggested called the Swin Transformer model [3]. In contrast to the Vision Transformer that processes the entire image at once by dividing it into fixed-size patches and applying global self-attention to all patches, the Swin transformer divides the image into smaller, non overlapping windows and applies local window-based self-attention (Figure 5(a)). This reduces the computational complexity because the number of interactions is limited to within a window rather than across the entire image. Additionally, the Swin transformer introduces a novel mechanism of shifting these windows in alternate layers, thus enabling cross-window connections. This helps in integrating local and global contextual information more effectively (Figure 5(b)). Finally, the fact that the Swin transformer starts with smaller windows and gradually merges

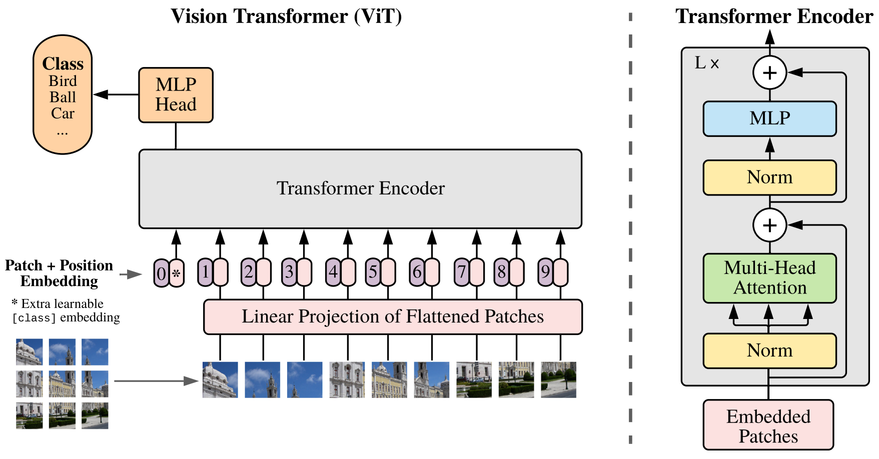
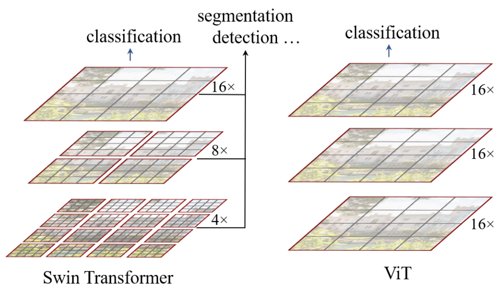
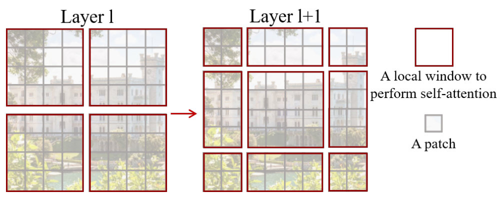


Figure 4: Model overview of a Vision Transformer [5]

them, reducing resolution while increasing the depth of features, enables a hierarchical architecture that processes features at multiple scales. This is beneficial for various vision tasks like object detection and segmentation, where multi-scale features are crucial. These advantages make the Swin transformer suitable as a general purpose backbone for various vision tasks and as will be explained later, we found this model to perform very well in our particular study.



(a) ViT vs Swin transformer



(b) The shifted window technique

Figure 5: Characteristics of the Swin transformer[3]

4 Methodology

Our code is made up of the following parts:
The **work.py** file sets up the training pipeline for our segmentation task. After loading all the necessary libraries, we use the function **parse_args()** to define and parse command-line arguments. Using this function we specify the model to be used in our experiment, as well as the model-specific parameters like learning rate, weight decay, batch size, number

of epochs, optimiser etc. The function can also check for which device to use (cuda, mps or cpu) depending on the platform that we run our code on and handles default and edge cases (like when no image size or batch size is specified). The function very usefully also offers a help option that will list and describe all the available options to the user.

The **main()** function orchestrates the execution flow. We initialize random seed for reproducibility, set up logging, and merge the default configuration with the command-line arguments using the parsed results. We initialise **wandb** for tracking our experiment and prepare the datasets and dataloaders for training and validation. We instantiate the model and move it to the designated device (CPU or GPU) and finally create a **Trainer** object with the model, dataloaders, and configuration and start the training process by instantiating the **Trainer()** class and calling the **train** method.

The **Trainer.py** performs several functions. It handles the logging of the best model's weights to wandb as 'Artifacts' for tracking and versioning. It performs a single training step by computing the model's output, calculating the loss using the defined loss criterion and updates the model weights based on the gradients. It also conducts an evaluation step, this time without gradient calculation, and updates the evaluation loss metric **'eval_losses'**. The training process is iterated over the number of epochs, during which we execute the training and validation steps, log the performance and check for improvement in validation loss in order to save the best model state. Finally we log the training and validation losses to wandb for monitoring.

The **model.py** file enables us to dynamically instantiate different models that are used for our analysis by mapping the model names stored on the **'models'** dictionary to the corresponding architectures. In the **CityScapesDataset.py** file we define the **CityScapesDataset()** class which handles the data loading for the CityScapes dataset. The **get_transforms()** function allows us to apply transformations to the images and masks loaded from the dataset. Due to limited computational resources, during most of our experiments, we first reduce the size of the images and masks (usually to 512×512px). For training data, additional random transformations (horizontal flipping and rotation) are applied for data augmentation to improve the robustness of the model. Images are then converted to a tensor with normalised pixel values between [0,1] and masks are converted to a long tensor from a numpy array, which is necessary for the categorical targets in our task.

The models directory contains the definitions of architectures of the various models we have tried for this analysis. We have started our investigation using a straightforward Fully Convolutional Network (FCN) model in **SimpleFCN.py**. The FCN is structured with an encoder-decoder design pattern. The encoder consists of three stages, each composed of a convolutional layer followed by a ReLU activation function and a max-pooling layer. This sequence allows us to process the image to extract basic features and reduces its spatial dimensions. In the decoder, we reverse the process of the encoder, using transposed convolutions to progressively reconstruct the spatial dimensions of the output. The final layer of the decoder increases the spatial dimensions back to the size of the original image and adjusts the depth to the number of output classes.

The second model we have deployed is the U-net model as defined in **UNetModel.py**.

Our U-net model consists of the following components: the **Block()** class defines a basic convolutional module that is made up of a convolutional layer followed by batch normalisation and a ReLU activation function. The **Downsampler** module is used to process and downsample the input tensor and is part of our encoder. It consists of two convolutional Block modules followed by a max pooling operation to reduce the spatial dimensions by half. The downsampling step increases the receptive field and allows the network to capture more abstract features at each level. The **Upsampler** module on the other hand is used as part of the decoder and is responsible for upsampling the feature maps and concatenating them from the corresponding downsampler through skip connections. It uses a transposed convolution to increase the spatial resolution and then processes the concatenated output through two Block modules. Finally, we construct our model by stacking together four Downsampler modules which progressively reduce the spatial dimensions and increase the depth of feature maps (64, 128,256,512 channels). Two Block modules process the deepest features without changing their dimensions. This is the "Bottleneck' part of the network and it bridges the encoding and decoding paths. The decoder is made up of four Upsampler modules which progressively increase the spatial dimensions and reduce the depth of the feature maps. Each Upsampler uses the output from the corresponding Downsampler for concatenation (skip connection), which helps in reconstructing the segmentation maps with detailed features. A final convolutional layer with kernel size 1×1 is used to map the 64 channels back to our number of output channels (classes), therefore providing our segmentation output.

5 Results

6 Conclusions

7 Reflections

References

[1] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. *CoRR*, abs/1604.01685, 2016.

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[3] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *CoRR*, abs/2103.14030, 2021.

[4] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.