Image Segmentation Project

PROJECT GITHUB LINK: mob9221/3ml3_project: 3ML3 Final Project - Image segnet (github.com)

Table of Contents

Important Note: A Jupyter notebook is submission is not possible for this project, please use the github link above to view the project code. I have also included the latest snapshot of the repository as a zip file. The entry point is in the main.py file!

Introduction

Project Objective

The purpose of this project will be to train a neural network to accurately segment and classify pixels in an image according to what class of object the pixel belongs to (e.g red for lane lines, green for obstacles) for autonomous driving. This is different from object detection since the differentiation between objects is not known (i.e no bounding boxes). An input image will be taken from a camera sensor, preprocessed to transform colour spaces if necessary, denoised and fed into the trained neural network to semantically segment the image. The training and validation dataset will be sourced from the open-source KITTI-360 dataset [1].

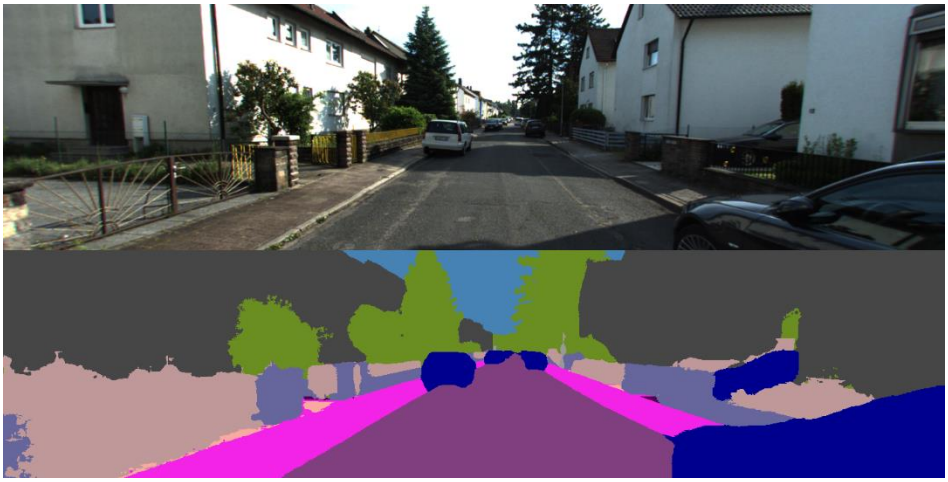

Figure 1 Raw and labelled image example from KITTI-360 training set

Project Motivation

Autonomous vehicles have become a very popular subject in recent times. The auto manufacturing company Tesla has been in the media spotlight recently showcasing the strengths and downsides of machine learning when it comes to solving the problem of autonomous driving. An array of sensors are usually used to scan the environment and determine obstacles and road markings to create a local map around the vehicle for the autonomous driving software to navigate in. The sensor suites of

autonomous vehicles usually include hardware such as cameras, radar, LiDAR and ultrasonic sensors with each of them serving a unique purpose. Radars, LiDARs and ultrasonics are useful for determining depth but are unable to provide meaningful semantic information about the object that is being detected. Likewise, cameras are useful for determining different types of obstacles and markings in an image but are not adept at predicting depth in an image. The system must have a robust image segmentation network to detect objects visually to fuse them with depth data from other sensors. For this project, we will only be handling image semantic segmentation.

Paper Outline

The initial sections of the paper will discuss the implementation of a model training pipeline to accomplish the task of accurate image segmentation from start to finish. This will involve selecting the data representation, loss function, network architecture and performance metrics. Detailed discussions and mathematical definitions will be provided for each section using equations and figures. The later sections will discuss the specifications of the dataset (image resolution, classes etc.), optimizers and the results from training the model using the pipeline (validation loss, training loss, accuracy metrics, outputs etc.).

Analysis of Training Pipeline Solutions

Image Representation

One of the hurdles with using images as inputs to a neural network is that they are not inherently understood by a neural network since they only work with numbers. This can be remedied using the fact that images are still stored numerically in memory using the corresponding RGB (Red, Green, Blue) value of every pixel traditionally in a 2d matrix data structure using arrays. There are many alternative colour spaces (colour value representations) that may be used to represent an image for

input into a neural network such as HSL, LUV and YUV. Research indicates that RGB is still one of the best colour spaces to use when training convolutional neural networks for image processing with LUV being a close second [2].

Loss Function

Once the image has been transformed into the colour space to be used for the neural network, the segmentation network will require a significant amount of training to produce meaningfully accurate results. The main process of training will require us to decide the cost/loss function that we wish to use for optimizing the neural network. The type of cost function can have significant impacts on the time required for training and the performance of the trained network. There are many cost functions to choose from such as the Mean Squared Error (MSE) cost function, the Categorical Cross-Entropy (CCE) cost function and the Dice co-efficient Loss function. The MSE loss function is not ideal for this project since we are not attempting to perform regression training.

The Dice Co-efficient is a useful metric for measuring the accuracy of the output as a whole but might not be well suited for this problem as it does not take individual classes into account. It is calculated as follows:

$$Dice\ Coefficient = \frac{2 * True\ Positives}{2 * True\ Positives + False\ Negatives + False\ Positives}$$

The dice co-efficient will calculate the overlap between the predicted image mask and the correct image mask, essentially performing intersection over union. This will be used as a secondary metric for evaluating model performance throughout the training process. The Dice loss function is simply calculated as:

$$Dice\ Loss = 1 - Dice\ coefficient$$

The value of the dice coefficient approaches 1 as the overlap between the predicted image mask and the true image mask increases [3]. Since this loss function does not account for different classes explicitly, it is better to use a loss function such as categorial cross entropy.

The image segmentation problem can alternatively be phrased as a per-pixel classification problem for which the Categorical Cross-Entropy loss function seems much more appealing since it accounts for discrete classes. The CCE loss function is defined as follows:

$$CCE = \frac{1}{M * N} \sum_{i=0}^{M} \sum_{j=0}^{N} \sum_{i=1}^{n} -T_i * \log(P_i) \qquad for\ n\ classes$$

The output of the neural network is a vector of size n containing likelihood scores for each class per pixel. This vector can be converted to a normalized probability distribution vector using the softmax function where all probabilities add up to 1. The softmax function is defined as follows:

$$Softmax(X) \frac{e^{X_i}}{\sum_{k=1}^{C} e^{X_k}} \quad for\ all\ X_i \in X$$

The third summation sums up the difference between the actual probability of the correct (1.0) and the output of the model (between 0 and 1) for the correct class. For example, if the network outputted a score vector for a pixel: [0.89, 0.01, 0.07, 0.03] where the label for that pixel is [1, 0, 0, 0], the value of T_i would be 1 for the first class only and everything else would be discarded since they will get multiplied by 0. This would lead to the sum: - [1 * log(0.89) + 0 * log(0.01) + 0 * log(0.07) + 0 * log(0.03)] = -log(0.89) = 0.05. This is calculated for every pixel in each row(M) and column(N) of the image and then averaged out of the number of pixels (1/M*N). As the function is logarithmic, it will penalize scores that are further away from 1 a lot more approaching infinity if the model outputs a probability of 0% for the correct class. Optimizing this loss function should yield a sufficiently performant model for this task.

Neural Network Architecture

This project requires image processing to be done within the neural network. An architecture based around convolution and pooling will be necessary to extract and learn features from an image. Convolution in the context of neural networks and image processing is illustrated in the following image:

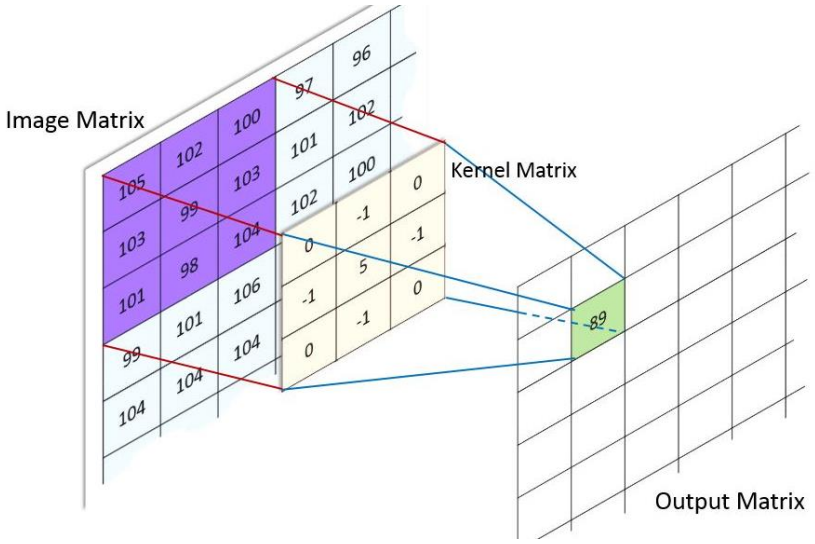

Figure 2 Depiction of the convolution operation

A kernel matrix is multiplied across successive rows and columns of the input image which outputs a feature map with a lower resolution. Generally, the convolution kernel is trained to differentiate specific features such as edges, lines etc. A higher numeric output corresponds to an increased chance of the presence of the feature that the kernel is attempting to extract. Running successive convolutions with the outputs from each daisy-chained to the next, powerful networks can be created for extracting useful features from images.

Pooling is another powerful technique employed throughout image processing networks. The concept is like convolution and involves a sliding kernel which extracts features from feature maps. Pooling is generally used on the outputs of convolutions to further down-sample the feature map and extract relevant information by taking the maximum value in the kernel's window (Max Pooling) or

averaging the values in the kernel's window (Average Pooling). It can also lead to less memory usage and faster model performance as dimensionality reduction happens with each pooling operation. Max and average pooling operations are illustrated in the following image:

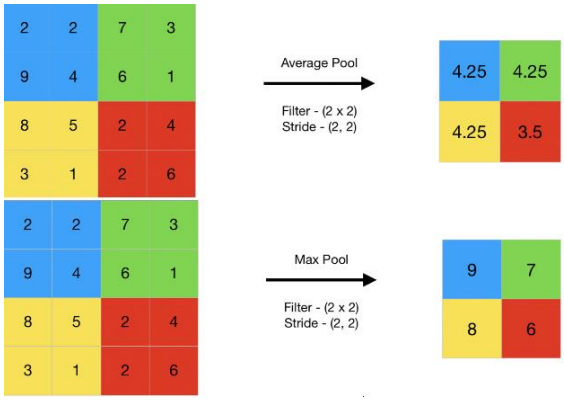

Figure 3 Max Pooling and Average Pooling

There are many heavily researched and proven architectures available to perform the task of per-pixel image segmentation, with some of the popular ones being U-Net, Resnet/ResGANet, Feature Pyramid Networks (FPN) and DeepLab (V1,V2,V3). U-Net with Fully Convolutional Networks has been proven to be a very efficient architecture to learn image segmentation, as it has won many awards in competitions where image segmentation was the goal such as the Kaggle competition using the Carvana dataset [4]. U-Net is also a great choice since it is relatively easy to understand using the concepts learned throughout the course and requires relatively little data to train and obtain decent results [5], compared to other architectures that can end up being massive in size, significantly increasing training time and computation power requirements. This is beneficial since the hardware available was a single desktop GPU which is inferior in comparison to the GPU clusters that are traditionally used to train large models. Implementing modules from scratch as much as possible was one of the goals of the project which also favoured the selection of U-Net since it is relatively easy to implement and has plentiful documentation available. The diagram of the U-Net architecture using Fully Convolutional Layers as illustrated by Ronneberger et. Al [5] for biomedical image segmentation is shown in the following figure:

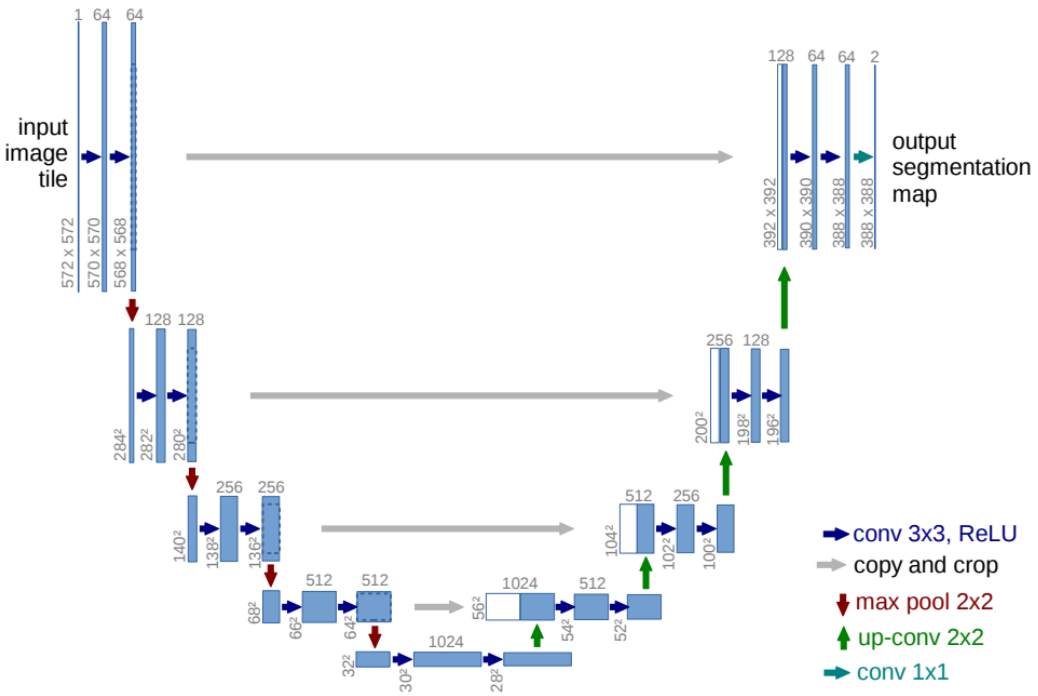

Figure 4 U-Net Architecture with convolution layers

At every level, double convolution operations are executed with the ReLU (Rectified Linear Unit) activation function. The RelU function acts as a linear pass-through filter for all positive real numbers which is helpful since none of the final or intermediate outputs is expected to be negative. The function is defined in the following equation:

$$ReLU(x) = \max(0, x)$$

The outputs of the convolutions and activation function are fed into a max-pool layer. Coming downwards from the left, the architecture encodes features and down-samples every feature map with each layer. At the very bottom, there is a “bottleneck” layer that down-samples even further which marks the point where the encoded image is generated for up-sampling. The right side of the figure depicts the up-sampling layers that attempt to decode features. Starting with the bottleneck layer, the output of each successive layer goes through an upsampling convolution which outputs a feature map that is twice the twice resolution of the input and decreases the number of features with each step. At

each layer of the encoding process, the output from the final convolution operation is saved and concatenated with the input to the corresponding decoding convolution layer on the right side which is shown by the grey arrows in the figure. This has been shown to help with the localization of objects in the image since the model can learn the relation between encoded features and corresponding up-sampled features. The output of the final convolution (1x1) is converted to probabilities using the softmax2d function. The output from the softmax function is used to calculate the categorical cross entropy loss function defined earlier. The number of output channels depicted in the figure is 2; This can be set to the number of possible classes in the dataset, outputting a probability score for every class per pixel.

This entire architecture is heavily inspired by the autoencoder-decoder concept, which is often applied for dimensionality reduction of datasets. In this case, the dimensionality reduction happens on hundreds of dimensions where the image is reduced from the RGB (W*H*3) color space to (W*H*1) assigning an integer class to each pixel.

Dataset Specifications

The KITTI-360 dataset is an open-source dataset published by the Autonomous Vision Group at the University of Tubingen in Germany [1]. The dataset contains the recording of camera and LiDAR sensors taken on many drives in Germany. The dataset contains 78,000 images and corresponding semantic label annotations with 45 possible classes. Each image has a resolution of 1408 x 376 pixels with 3 color channels (RGB). The images are wide in resolution due to being captured from a stereo camera which is designed for a wide field of view. For this project, the image will be downsampled to a half resolution of 704 x 178 pixels to decrease memory usage and speed up training/inferencing times. The publishers of the dataset provide the following distribution of labels over the 78,000 images:

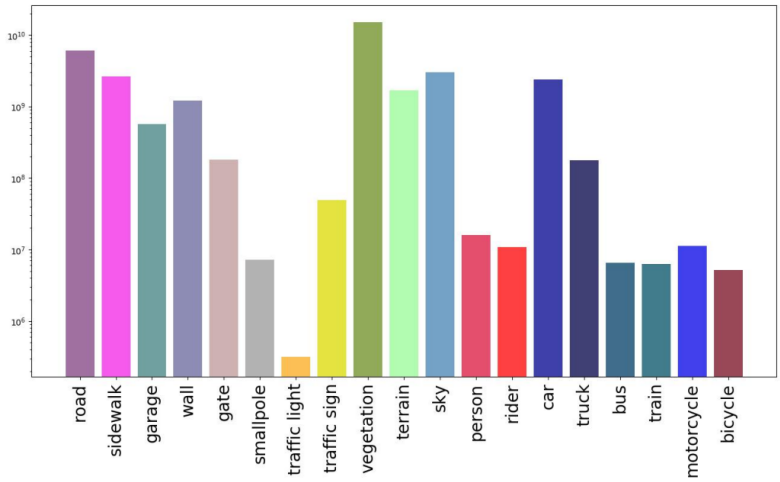

Figure 5 Distribution of 2D semantic labels of 78k frames

The distribution of labels is not uniform across classes. Classes such as vegetation, roads, sidewalks, and sky dominate the dataset [1]. This bias in the dataset will need to be accounted for in the model's accuracy metrics since a regular accuracy score will not be sufficient to correctly determine model performance. A balanced accuracy score will be used alongside the dice coefficient mentioned previously. The balanced accuracy score takes the number of classes into account by calculating the model's recall accuracy per class and averaging the score over the number of classes. Balanced accuracy can be calculated using the following equation:

$$Balanced\ Accuracy = \frac{\sum_1^C \frac{TP}{TP+FN}}{C} \quad where\ TP = True\ Positives, FN = False\ Negatives$$

To measure the model's performance, the balanced accuracy score will only be calculated using specific classes deemed safety critical for self-driving purposes: Cars, Trucks, Busses, Motorcycles, Bicycles, Persons, Road. The accuracy of the model on other classes is not relevant to this application, thus will not be prioritized during performance analysis. The traffic light class is also important for autonomous driving but will not be considered in this project since the labels are quite sparse in quantity.

For training purposes, the entire dataset will not be used. The model will be trained on 7,000 images using small batch sizes ranging from 2-8 depending on the memory available on the training hardware. As U-Net is an efficient architecture, this should still yield interesting and decently accurate results for the important classes in theory.

Learning Optimizer

Training the U-Net model using the KITTI-360 dataset will require an optimizer to minimize the categorical cross entropy loss function with successive epochs of training. Two of the popular optimizers commonly used are Stochastic Gradient Descent (SGD) and the Adaptive Moment Estimation (Adam) optimizers. The Adam optimizer combines the advantages of the adaptive gradient algorithm which improves performance on problems with vanishing gradients and Root Mean Square Propagation which performs exponential averaging of the recent gradient values along with normalization of the gradient [6]. Additionally, the Adam optimizer performs exponential averaging of the gradient magnitude. This process can be described using the following equations:

$$\vec{d}^{k-1} = \beta * \vec{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\vec{w}^{k-1}\right)\right) \quad Exponential\ averaging\ of\ direction$$

$$h^{k-1} = \varphi * h^{k-2} + (1-\varphi)\left(-\nabla g\left(\vec{w}^{k-1}\right)\right)^2 \quad Exponential\ averaging\ of\ magnitude$$

$$\vec{w}^{k} = \vec{w}^{k-1} - \alpha * \frac{\vec{d}^{k-1}}{\sqrt{h^{k-1}}} \qquad Weight\ update\ per\ learning\ step$$

The Adam optimizer has been proven to be significantly more effective on computer vision problems compared to SGD since the gradients have a higher tendency to vanish during backpropagation in the upper layers of the model, especially if the model has numerous layers [6]. All these problems will be present in this project as well, making Adam Optimizer the ideal choice to optimize the loss function.

Supplemental Techniques

Image Augmentation: Image augmentation is the process of performing random transformations on images to increase the effectiveness of the dataset during training [7]. The transformations applied usually applied include rotation, vertical flips and horizontal flips. The Albumentations library provides these functions along with PyTorch integration. The following transform setup was used for training images:

```
train_transform = A.Compose(
    [
        A.Resize(width=IMAGE_WIDTH, height=IMAGE_HEIGHT), # Resize
to 704x188
        A.Rotate(limit=35, p=1.0), # Rotate by 35 degrees, Always
happens
        A.HorizontalFlip(p=0.5), # Flip horizontally, 50% chance
        A.VerticalFlip(p=0.1), # Flip vertically, 10% chance
        ToTensorV2() # Convert to PyTorch tensor
    ],
```

GPU and Parallel GPU Acceleration: PyTorch and other frameworks have built-in support for GPU processing acceleration libraries such as CUDA from Nvidia. Most operations involving neural network training and inferencing can be heavily parallelized, making GPUs ideal for the task. GPUs are heavily optimized for parallel computing tasks since most of the traditional graphics pipelines are designed to exploit parallel computing. This can be extended to multiple GPUs running on the same system as well in "clusters". PyTorch makes it very easy to parallelize computing over multiple GPUs. The model just needs to be wrapped in a nn.DataParallel module and the batch size needs to be increased:

```
model = Model(arg)
model = torch.nn.DataParallel(model, device_ids=[0, 1]) # 2 GPUs
model.to(device)
```

These supplemental techniques combined can yield significant improvements in training speed and resulting model performance.

Training and Evaluation Results

7000 images were used from the dataset during training and validation with an 80/20 split. The training was initially conducted on a desktop PC with an Nvidia RTX 1070Ti GPU. Training a single epoch with a batch size of 2 took 27 minutes per epoch. Due to this limitation, only 20 epochs were trained and the validation and training loss decreasing with every epoch. Future training was conducted on the McMaster CAS GPU clusters equipped with dual RTX 3090Ti GPUs bringing the single epoch training time to 90 seconds. 50 epochs were trained given this advantage and the results remained largely the same. The plot of validation and training loss vs the number of epochs is shown below:

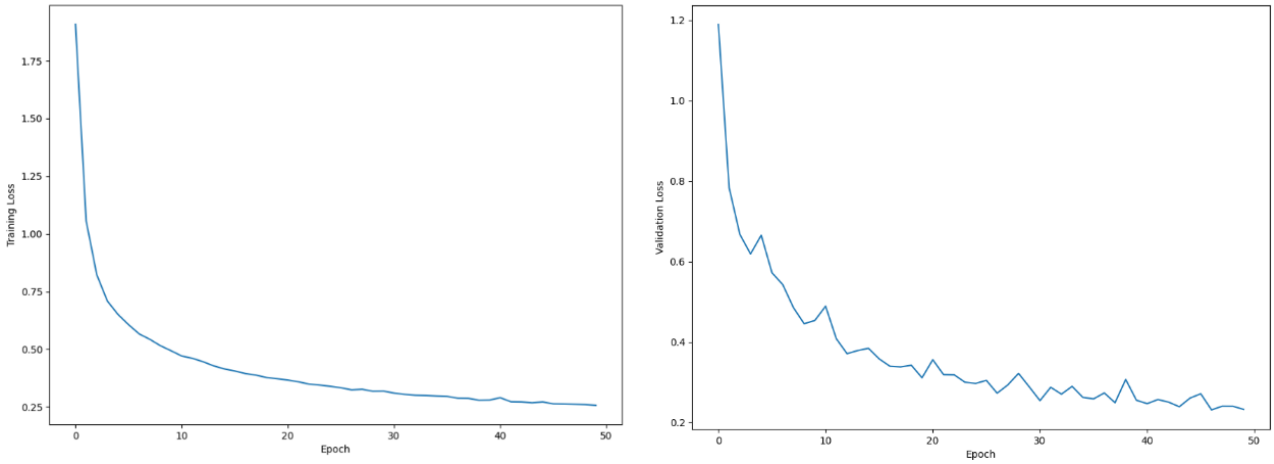

Figure 6 Training Loss vs Epochs (Left), Validation Loss vs Epochs (Right)

We can observe the validation and training loss generally decrease over time. There are small fluctuations in the validation loss plot, which is indicative of slight overfitting happening occasionally that is generally corrected in the next few epochs. Considering the plots are generally decreasing over time, leads to the conclusion that there could be further improvements made if more training was conducted. There are no signs of obvious overfitting based on the validation loss plot either. Overfitting would be indicated by a constantly increasing validation loss plot which is not observed here. Initially, it was expected that the model's data learning capacity was reaching its limit as the derivative of the loss

function compared to the number of epochs was steadily approaching 0. An additional 150 epochs of training were executed on the GPU cluster to test the hypothesis. After 12 hours of additional training, the results are shown in the following figure:

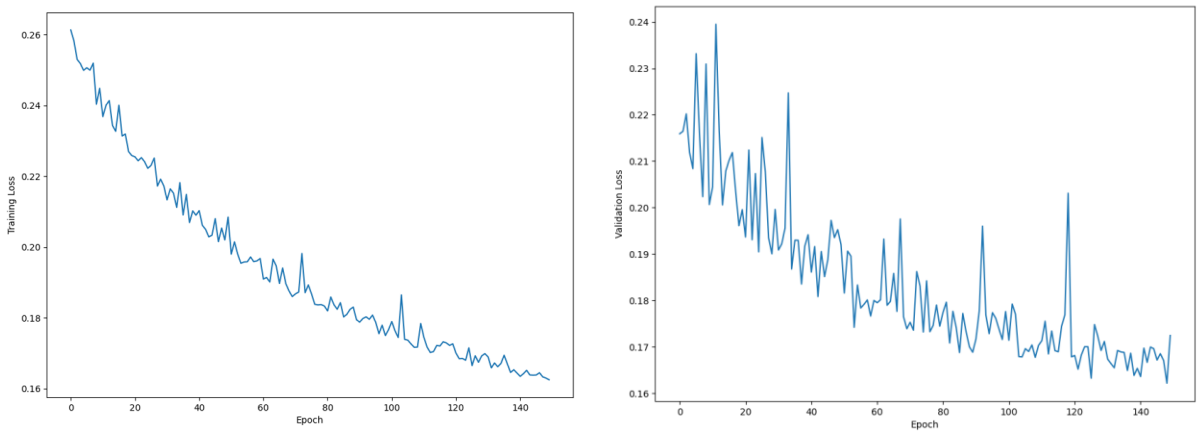

Figure 7 Training and Validation loss vs 150 epochs of additional training

It is important to note that the figures above show the loss functions with an additional 50 epochs of training performed in the previous training session. The previously trained model was loaded up along with the prior training data associated with it using PyTorch to continue training. It is simultaneously surprising and impressive that the model could still learn more based on the limited set of images being provided to it. It is also interesting to note that the validation loss function fluctuated a lot more during this training session but still generally decreased over time. This could be the result of a different set of validation images being picked at random every time a new training session is started. Based on the 200 epochs of training, it can be sufficiently concluded that the model has a decent learning capacity and could probably be trained further for better results. Ideally, the model would be trained on the entire 78,000 images in the dataset but memory, hardware and time constraints prevent this. In a real-world scenario, any team or company working on such a project would have access to even more data and computing power, allowing them to train more complex and powerful models compared to U-Net.

As the project objective relates to autonomous vehicles, safety is a top concern. The model needs to have a very high true positive and low false positive rate (recall). This must hold especially for the most important safety-critical object classes such as cars, pedestrians, and cyclists. The following accuracy scores based on a random sample of 1200 images, were calculated for the model iteration that achieved the lowest validation loss score:

Class	Precision	Recall	# of samples
Car	0.978	0.987	22,929,424
Truck	0.966	0.922	555,484
Bus	0.4707	0.011	37,354
Motorcycle	0.903	0.8985	146,162
Bicycle	0.6264	0.675	34,393
Person	0.77	0.3575	195,388
Road	0.987	0.989	29,806,204

Figure 8 Accuracy metrics for every important class

Average Dice Score = 0.701

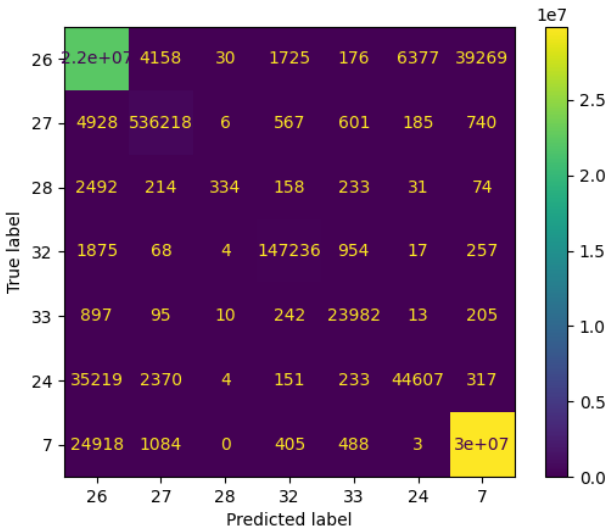

Figure 9 Confusion matrix with important classes: 26-Car, 27-Truck, 28-Bus, 32-Motorcycle, 33-Bicycle, 24-Person, 7-Road

The dice score indicates the overlap between the predicted image masks and the actual image masks as discussed in an earlier section. A dice score of 1.0 indicates perfect overlap and a score of 0.701 indicates a 70.1% overlap of the predictions with the true image masks. This metric is useful as a general performance metric but does not account for class wise accuracy, for which the recall and precision scores are needed.

The precision score indicates the total number of correctly identified relevant instances as a percentage of all instances classified as that class. The recall score indicates the number of correctly identified instances as a percentage of the total number of instances that should have been correctly identified:

$$Precision = \frac{True\ Positive}{True\ Positives + False\ Positives} \qquad Recall = \frac{True\ Positives}{True\ Positive + False\ Negatives}$$

Ideally, for this safety-critical application, one would want both of these metrics to be at their highest value of 1.0, which is not practical in most cases. As observed in the table, the number of samples for the specific class is highly correlated with the precision and recall scores. Classes such as Bus and Bicycle suffer from a lack of examples to learn from compared to the Road and Car classes. Additional training examples of buses and bicycles should help the model learn how to classify them better. It is important to note that these examples are per pixel (i.e 37,354 bus examples mean 37,354 pixels in all pictures combined belonging to the bus class). These results are also consistent with the class label distribution discussed in Figure 5. These classes were prioritized on purpose compared to the rest since they represent the most critical objects that a self-driving vehicle must detect. The precision and recall for the car, truck, motorcycle and road classes are ideal for the autonomous driving application since they are close to 100%. The model will need to be retrained with more samples of busses, bicyclists and pedestrians (persons) before it could be deployed onto a real vehicle since the recall and precision

of below 50% poses a genuine safety risk. Many of these results are neatly summarized in the confusion matrix in Figure 9.

The following figure shows a sample prediction from the trained model:

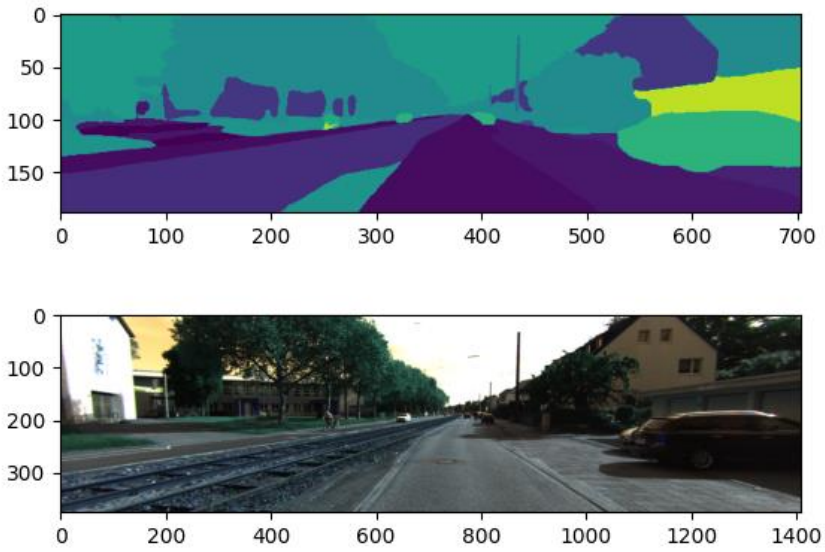

Figure 10 Sample prediction from the trained model. Prediction mask on the top, raw image on the bottom

Every pixel in the output mask is colored a specific color related to the class number. It is impressive to see the performance of the model even on smaller vehicles that are further away in the image.

Conclusion

The objective of the project was to train an accurate image segmentation model for use in autonomous vehicles since cameras can provide useful semantic information to supplement information from other depth sensors such as radars. The safety-critical nature of the application necessitated a high-performance standard to minimize dangerous incidents involving self-driving vehicles. The process behind the development of each part of the model training pipeline was discussed, beginning with the data format selection which was selected to be the RGB color space based on research indicating that it

yielded the best performance during training. The loss function was compared between the Dice loss function and the Cross Categorical Entropy loss function based on the versatility of CCE in multi-class prediction applications. The U-Net model architecture was selected with Fully Convolutional Networks as the backbone due to its proven efficiency and ability to obtain decent accuracy scores with very little data. The individual components and supplemental techniques such as image augmentation and GPU accelerated training were set up in the pipeline training infrastructure which was developed in PyTorch. Once the model was trained using PyTorch on the KITTI-360 dataset, the results and metrics from 200 epochs of training were discussed. The resulting model performed well on many object classes such as cars, trucks, roads and motorcycles achieving an accuracy score of over 90% for each of them reaching as high as 98.9% for roads. The model suffered from a lack of training examples for classes such as bicycles and buses. This could be remedied by increasing the size of the training set from the 7,000 images used to the full dataset consisting of 78,000 images but was not possible for this project due to hardware and time constraints.

Overall, the project yielded interesting results and was very ambitious in its implementation. The training pipeline was developed over 150 hours of development and debugging time and an additional 50 hours of training time.

References

[1] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The KITTI dataset,” The International Journal of Robotics Research, vol. 32, no. 11, pp. 1231–1237, Sep. 2013, doi: 10.1177/0278364913491297.

[2] K. S. Reddy, U. Singh, and P. K. Uttam, “Effect of image colourspace on performance of convolution neural networks,” in 2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), May 2017, pp. 2001–2005. doi: 10.1109/RTEICT.2017.8256949.

[3] C. H. Sudre, W. Li, T. Vercauteren, S. Ourselin, and M. Jorge Cardoso, “Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations,” in Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support, vol. 10553, M. J. Cardoso, T. Arbel, G. Carneiro, T. Syeda-Mahmood, J. M. R. S. Tavares, M. Moradi, A. Bradley, H. Greenspan, J. P. Papa, A. Madabhushi, J. C. Nascimento, J. S. Cardoso, V. Belagiannis, and Z. Lu, Eds. Cham: Springer International Publishing, 2017, pp. 240–248. doi: 10.1007/978-3-319-67558-9_28.

[4] K. Team, “Carvana Image Masking Challenge–1st Place Winner’s Interview,” *Kaggle Blog*, Dec. 02, 2019. https://medium.com/kaggle-blog/carvana-image-masking-challenge-1st-place-winners-interview-78fcc5c887a8 (accessed Apr. 20, 2023).

[5] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” 2015, doi: 10.48550/ARXIV.1505.04597.

[6] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” 2014, doi: 10.48550/ARXIV.1412.6980.

[7] S. Yang, W. Xiao, M. Zhang, S. Guo, J. Zhao, and F. Shen, “Image Data Augmentation for Deep Learning: A Survey,” 2022, doi: 10.48550/ARXIV.2204.08610.