

# Bonnet: An Open-Source Training and Deployment Framework for Semantic Segmentation in Robotics using CNNs

Andres Milioto

Cyrill Stachniss

**Abstract**—The ability to interpret a scene is an important capability for a robot that is supposed to interact with its environment. The knowledge of *what* is in front of the robot is, for example, key to navigation, manipulation, or planning. Semantic segmentation labels each pixel of an image with a class label and thus provides a detailed semantic annotation of the surroundings to the robot. Convolutional neural networks (CNNs) became popular methods for addressing this type of problem. The available software for training and the integration of CNNs in real robots, however, is quite fragmented and difficult to use for non-experts, despite the availability of several high-quality open-source frameworks for neural network implementation and training. In this paper, we propose a novel framework called Bonnet, which addresses this fragmentation problem. It provides a modular approach to simplify the training of a semantic segmentation CNN independently of the used dataset and the intended task. Furthermore, we also address the deployment on a real robotic platform. Thus, we do not propose a new CNN approach in this paper. Instead, we provide a stable and easy-to-use tool to make this technology more approachable in the context of autonomous systems. In this sense, we aim at closing a gap between computer vision research and its use in robotics research. We provide an open-source framework for training and deployment. The training interface is implemented in Python using TensorFlow and the deployment interface provides a C++ library that can be easily integrated in an existing robotics codebase, a ROS node, and two standalone applications for label prediction in images and videos.

## I. INTRODUCTION

Perception is an essential building block of most robots. Autonomous systems need the capability to analyze their surroundings in order to safely and efficiently interact with the world. Augmenting the robot's camera data with the semantic categories of the objects present in the scene, has the potential to aid localization [2, 3, 22], mapping [16, 29], path planning and navigation [10, 30], manipulation [5, 27], precision farming [19, 20] as well as many other tasks and robotic applications. Semantic segmentation provides a pixel-accurate category mask for a camera image or image stream. The fact that each pixel in the images is mapped to a semantic class, allows the robot to obtain a detailed semantic view of the world around it and aids to the understanding the scene.

Most methods, which represent the current state of the art in semantic segmentation, are based on fully convolutional neural networks. The success of neural networks for many tasks from machine vision to natural language processing has triggered the availability of many open-source development and training frameworks such as TensorFlow [1]

All authors are with the University of Bonn, Germany. This work has partly been supported by the EC under the grant number H2020-ICT-644227-Flourish.

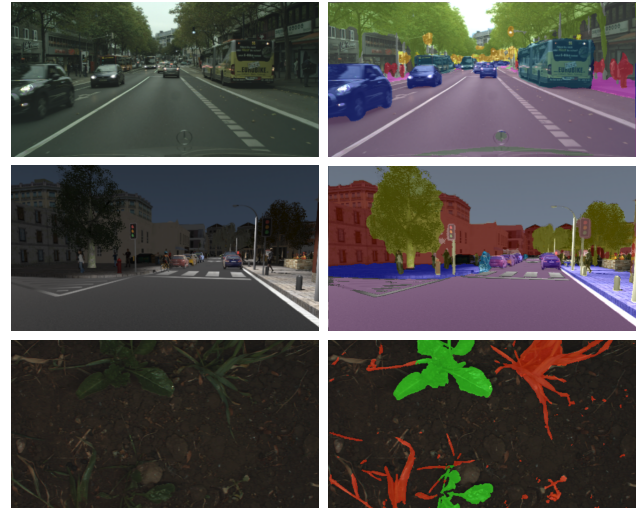


Fig. 1. Sample predictions from our framework. Left: Raw RGB images. Right: Overlay with semantic segmentation label from CNN prediction. From top to bottom: Cityscapes dataset [9], Synthia dataset [25], Crop-Weed agricultural dataset [6]. Best viewed in color.

or Caffe [15]. Even though these tools have simplified the development of new networks and the exploitation of GPUs dramatically, they are still non-trivial to use for a novice. Companies such as NVIDIA and Intel have furthermore developed custom accelerators such as TensorRT or the Neural Compute SDK. Both use graphs created with TensorFlow or Caffe as inputs and transform them into a format in which inference can be accelerated by their custom hardware. As with the other frameworks, their learning curve can be steep for a developer that actually aims at solving a robotics problem but which relies on the semantic understanding of the environment. Last but not least, source code of computer vision research related to semantic segmentation is often made available, which is a great achievement. Each research group, however, uses a different framework and adapting the trained networks to an (own) robotics codebase can sometimes take a considerable amount of development time.

Therefore, we see the need for a framework that allows a developer to easily train and deploy semantic segmentation networks for robotics. Such a framework should allow developers to easily add new research approaches into the robotic system while avoiding the effort of re-implementing them from scratch or modifying the available code until it becomes at least marginally usable for the research purpose. This is something that we experienced ourselves and observed in the community too often.

The contribution of this paper is a novel software framework that provides a modular implementation of semantic segmentation using CNNs. It solves training and deployment on a robot. Thus, we do not propose a new CNN approach here. Instead, we provide a set of tools to make this technology easily usable in robotics and to enable a larger number of people to use CNNs for semantic segmentation on their robots. We strongly believe that our framework allows the scientific robotics community to save time on the CNN implementations, enabling researchers to spend more time to focus on how such approaches can aid robot perception, localization, mapping, path planning, obstacle avoidance, manipulation, safe navigation, etc. Our framework relies on TensorFlow for our graph definition and training, but provide the possibility of using different backends with a clean and stable C++ API for deployment. It allows for the possibility to transparently exploit custom hardware accelerators that become commercially available, without modifying the robotics codebase.

In sum, our software framework provides (i) a modular implementation platform for training and deploying semantic segmentation CNNs in robots; (ii) a sample architecture that performs well for a variety of perception problems in robotics while working roughly at sensor framerate; (iii) a stable, easy to use, C++ API that also allows for the addition of new hardware accelerators as they become available; (iv) a way to promptly exploit new datasets and network architectures as they become available in the computer vision and robotics community.

Although we do not propose a new scientific method, we believe that this work has the potential to have a strong positive impact on the robotics community. Our software is available as open-source at <https://github.com/Photogrammetry-Robotics-Bonn/bonnet>. Furthermore, we make this paper available on arXiv to promote the use of our framework.

## II. RELATED WORK

Semantic segmentation is important in robotics. The pixel-wise prediction of labels can be precisely mapped to objects in the environment and thus allowing the autonomous system to build a high resolution semantic map of its surroundings.

One of the pioneers in efficient feed-forward encoder-decoder approaches to semantic segmentation is Segnet [4]. It uses an encoder based on VGG16 [28], and a symmetric decoder outputting a semantic label for each pixel of the input image. The decoder uses the encoder pooling indexes to perform the unpooling to recover some of the lost spatial resolution during pooling. Segnet is available as a Caffe implementation and has pre-trained weights for several datasets. U-Net [24], which was released contemporaneously, exploits the same encoder-decoder architecture but uses a decoder concatenation of the whole encoder feature map instead of sharing pooling indexes. This allows for more accurate decision boundaries, which comes at a higher computational and memory cost. U-Net is available as an implementation in a modified Caffe version and provides pre-trained weights for

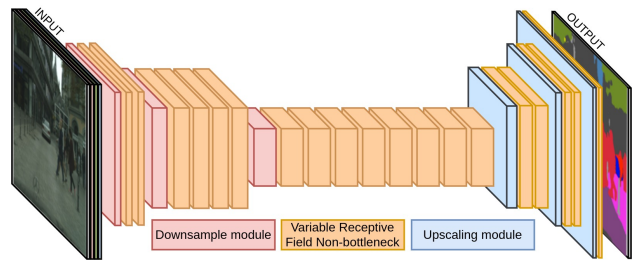


Fig. 2. Example of an encoder-decoder semantic segmentation CNN, based on the non-bottleneck idea behind ERFNet [23]. Best viewed in color.

a medical dataset. PSP-Net [32], the current state of the art in semantic segmentation, uses ResNet [12] as the encoder, and exploits global information through a pyramid of average-pooling layers after the latter, to provide more accurate semantics based on the environment of the image objects. PSP-Net is also available as a modified Caffe implementation and comes with pre-trained weights from different scene parsing datasets. All of these architectures are based on encoders such as VGG and ResNet, which focus on accuracy of the predictions rather than the execution speed for a near real-time application in robotics.

Other architectures use post-processing steps to improve the decision boundaries in the segmented masks. Some versions of DeepLab [7] use fully connected conditional random fields (CRF) in addition to the last layer CNN features in order to improve the localization performance. CRF-as-RNN [33] replaces the CRF with a recurrent neural network for prediction refinement, also deviating from a fully feed-forward implementation. Both approaches provide modified implementations of Caffe and pre-trained weights for some scene parsing datasets. Because of rather inefficient feature extractors and the post-processing steps, their execution speed is quite far away from the frame-rate of a regular camera, even when executed on the most powerful acceleration hardware available today.

Robots, however, need online inference capabilities for most applications. There has been work focusing on inference efficiency, both in terms of execution time and model size. Enet [21] proposes efficient down-sampling modules, efficient bottlenecks, and dilated convolutions to decrease the model size and to improve the computational efficiency. Enet is available as an implementation based on the scientific computing framework Torch and provides pre-trained weights. ICNet [31] proposes a compressed pyramid scene parsing network using an image cascade that incorporates multi-resolution branches to provide a more efficient implementation of PSP-Net that can run closer to real-time. It is available as a Caffe implementation based in PSP-Net, and contains pre-trained weights. ERFNet [23] proposes a way of widening each layer by replacing the bottleneck modules with efficient dilated separable convolution modules. It is available both, as Torch and PyTorch implementations, and contains pre-trained weights. Mobilenets-v2 [26] proposes inverted residuals and linear bottlenecks to achieve near

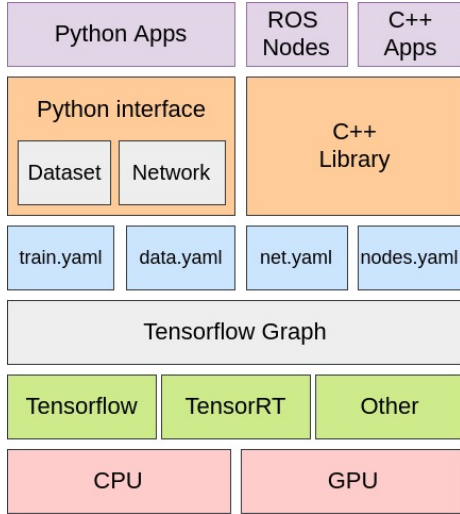


Fig. 3. Layout of the framework. Python interface is used for training and graph definition, and C++ library can use a trained graph and infer semantic segmentation in any running application, either linking it or by using the ROS node. Both interfaces communicate through the four configuration files in *yaml* format and the trained model weights.

state-of-the-art performance in semantic segmentation using efficient constrained networks. Mobilenets-v2 is available as a TensorFlow implementation.

This fragmentation of different systems and backends motivates our idea of providing a modular implementation framework, in which such architectures can be realized.

### III. BONNET: TRAINING AND DEPLOYMENT FOR SEMANTIC SEGMENTATION IN ROBOTICS

We provide our semantic segmentation framework called Bonnet with a Python training pipeline and a C++ deployment library. The C++ deployment library can be used standalone or as a ROS node. We provide a CNN implementation based on ERFNet, as depicted in Fig. 2 as well as pre-trained weights on three different scene parsing datasets. Our framework allows for fast multi-GPU training, for easy addition of new state-of-the-art architectures and available datasets, for easy training, retraining, and deployment in a robotic system. It furthermore allows for transparently using different backends for hardware accelerators as they become available. This all comes with a stable C++ API.

The usage of our framework is split in two steps. First, training the models to infer the pixel-accurate semantic classes from a specific dataset through a Python interface which is able to access the full-fledged API provided by TensorFlow for neural network training. Second, deploying the model in an actual robotic platform through a C++ interface which allows the user to infer from the trained model in either an existing C++ application or a ROS-enabled robot. Fig. 3 shows a modular description of this division, from the application level to the hardware level, which we explain in detail in the following sections.

### IV. BONNET TRAINING

The training of the models is performed through the methods defined through the abstract classes *Dataset* and *Network* (see Fig. 3), which handle the pre-fetching, randomization, and pre-processing of the images and labels, and the supervised training of the CNNs, respectively.

In order to train a model using our framework, there is a sequence of well-defined steps that need to be performed, which are:

- Dataset definition, which is optional if the dataset is provided in one of our defined standard dataset formats.
- Network definition, which is also optional if the provided architecture fits the needs of the addressed semantic segmentation task.
- Hyper-parameter tuning.
- GPU training, either through single or multi GPU. This step can be performed either from scratch, or from a provided pre-trained model.
- Graph freezing for deployment, which optimizes the models to strip them from training operations and outputs a different optimized model format for each supported hardware family.

#### A. Dataset Definition

The abstract class *Dataset* provides a standard way to access dataset files, given a desired split for it in training, validation, and testing sets. The codebase contains a general dataset parser which can be used to import a directory containing images and labels that are already split and put them in the standard dataset format, but the script can also be used as a baseline to implement it in a different way, given a different organization of the dataset files. The definition of each numeric label's semantic class, the colors for the debugging masks, the desired image inference size, and the location of the dataset are meant to be performed in the corresponding dataset's *data.yaml* file, of which there are several examples in the codebase. Once the dataset is parsed into the standard format, the abstract class *Network* knows how to communicate with it in order to handle the training and inference of the model. Besides the handling of the file opening and feeding to the CNN trainer, the abstract dataset handler performs the desired dataset augmentation, such as flips, rotations, shears, stretches, and gamma modifications. The dataset handler runs on a thread different from the training, such that there is always an augmented batch available in RAM for the network to use, but also allows the program to use big datasets in workstations with limited memory. The selection of this cache size allows for speed vs. memory adjustment which depend on the system available to the trainer.

#### B. Network Definition

Once the dataset is properly parsed into the standard format, the architecture has to be defined. We provide a sample architecture, which is designed with real-time operation in mind, and provide pre-trained weights for different datasets, and different network sizes, depending on the complexity

of the problem. Other network architectures can be easily added, given the modular structure of the codebase, and it is the main purpose of the framework to allow the implementation of new architectures as they become available. The abstract class `Network` (see Fig. 3) contains the definition of the training method that handles the optimization through stochastic gradient descent, inference methods to test the results, metrics for performance assessment, and the graph definition method, which each architecture overloads in order to define different models. Each architecture inherits the abstract class `Network` and redefines the graph in order to change the model type, which allows to easily add new architectures. If a new architecture requires a new metric, or a different optimizer, these can be modified simply by overloading its corresponding method of the abstract class. The interface with the model architecture is done through the *net.yaml* configuration file, which includes the selection of the architecture, the number of layers, number of kernels per layer, and some other architecture dependent hyper-parameters such as the amount of dropout [13], and the batch normalization [14] decay.

The interface with the optimization is done through the *train.yaml* file, which contains all training hyper-parameters, such as learn rate, learn rate decay, batch size, the number of GPUs to use, and some other parameters such as the possibility to periodically save image predictions for debugging, and summaries of the weights and activations histograms, which take a lot of disk space during training, and are only useful to have during hyper-parameter selection. There are examples of these configuration files provided for the included architectures in the codebase.

It is important to notice that since the abstract classes `Network` and `Dataset` handle most cases well with their default implementation, no coding is required to add a new task and train a model unless a number of very special cases. However, if a complex dataset is to be added, or a new network implementation is desired, the framework allows for its easy implementation.

### C. Hyper-parameter Selection

Once the network and the dataset have been properly defined, the hyper-parameters need to be tuned. We recommend doing the hyper-parameter selection through random-search, as single GPU jobs, which can be performed by starting the training with different configuration files (*net.yaml*, *train.yaml*), with all summary options enabled, and then choosing the best performing model for a final multi-GPU training until convergence. The framework is designed in this way for more simplicity, and because in this way the hyper-parameter selection jobs can be scheduled easily with an external job-scheduling tool. The hyper-parameters that can be configured are the following:

- Cache size: number of images and labels to keep in main memory by the dataset handler, ready for feeding to the training. This does not affect the performance of the model, but a higher number of fetched examples will train faster, avoiding reading directly from disk.

- Image properties: data augmentation techniques such as rotations, flips, shears, stretches, gamma modifications. The image size can also be treated as a hyper-parameter, trying to adjust it to the lower possible resolution as long as the performance is acceptable for the task. Since each application may require different levels of pixel accuracy, in speed and resource constrained applications such as robotics we want to avoid extra calculations, and thus the selected input image size is critical to runtime.
- Network properties: number of layers; size of kernels; number of kernels; batch normalization [14] decay rates and regularizer rates such as dropout [13] and weight decay.
- Training properties: Adam optimizer [17] learning rate and decays; type of weighting policy for dealing with unbalanced classes in the dataset and  $\gamma$  for focal loss [18]; batch size and number of GPUs.

A key factor for training that is worth mentioning is what to do when the pixel content is unbalanced for each class. If some of the classes of the dataset have a lower occurrence frequency, we need to address the problem properly to obtain a good performance. There are currently three different approaches to handle this type of problem implemented in our framework, considering as a starting point a pixel-wise vanilla log-loss for each image of the type:

$$L_{\text{vanilla}}(y, p) = - \sum_{i=1}^P \sum_{j=1}^C y_{ij} \log(p_{ij})$$

$$\text{where } p_j = \frac{e^{z_j}}{\sum_{j=1}^C e^{z_j}} \text{ for } j = 1, \dots, C$$

Here,  $C$  is the number of classes,  $P$  is the total number of pixels in the image,  $y$  is the ground truth value as a one-hot representation,  $z$  represents the pixel-wise un-bounded logits, and  $p$  its corresponding softmax value.

The first two methods incur into adding a per-class weight in the pixel-wise loss, resulting in a per-image loss and weighting strategies of the form:

$$L_{\text{weighted}}(y, p) = - \sum_{i=1}^P \sum_{j=1}^C w_j y_{ij} \log(p_{ij})$$

$$\text{where } w_j = \begin{cases} \frac{\tilde{f}_j}{f_j} & \text{for median frequency balancing} \\ \frac{1}{\ln(f_j + \epsilon)} & \text{for inverse log frequency balancing} \end{cases}$$

$$\text{and } f_j = \frac{P_j}{P} \text{ for } j = 1, \dots, C$$

In the case of “inverse log frequency balancing”,  $\epsilon$  needs to be set so that each  $w_j$  doesn’t surpass certain value for numerical stability (e.g., 10, or 50, also a hyper-parameter).

The third method was introduced in the context of object detection by RetinaNet [18], and penalizes the hard examples by using a “focal” loss function of the form:

$$L_{\text{focal}}(y, p) = - \sum_{i=1}^P \sum_{j=1}^C y_{ij} (1 - p_{ij})^\gamma \log(p_{ij})$$



These three methods address the problem of class balancing in different ways, and the weighting and focal loss can be combined in our framework by using a combined loss, with two tunable parameters  $\gamma$  and  $\epsilon$  of the form:

$$L_{\text{combined}}(y, p) = - \sum_{i=1}^P \sum_{j=1}^C w_{ij} y_{ij} (1 - p_{ij})^{\gamma} \log(p_{ij})$$

Most frameworks require the setting of the weight vector manually, but our framework calculates the frequencies automatically from the training set labels, and allows to select the strategy in the training configuration file *train.yaml*. For more information, please refer to the *readme* file within the code.

#### D. Multi GPU training

Once the most promising model is found, the training can be done with this hyper-parameter set using multiple GPUs to be able to increase the batch size, and hence, the speed of training. Changing the number of GPUs used for training is as simple as changing the setting in the *train.yaml* configuration file, but we recommend scaling the hyper-parameter set found following the procedure described in [11] for better results. The multi-GPU training, as described in Fig. 4, is performed by synchronously averaging the gradients obtained by a single SGD step in each GPU. For this, all model parameter are stored in main memory, and they are transferred to each GPU after each step of averaged gradient update. This is handled by the abstract network’s training method, and it is transparent to the user. The accuracy and Jaccard index (IoU) are periodically reported and the best performing models in the validation set are stored. We store both the best accuracy and the best intersection over union model, for posterior use in deployment. The mean Jaccard index (IoU) is used for the final evaluation:

$$mIoU = \frac{1}{C} \sum_{i=1}^C \frac{TP_j}{TP_j + FP_j + FN_j}$$

Another important work to make GPU training more efficient is the introduction of the concept of “checkpointed gradients” [8], which allows to fit big models in GPU memory in sub-linear space. This is done by checkpointing nodes in the computation graph defined by the model, and recomputing the parts of the graph in between those nodes during backpropagation. This makes it possible to calculate the network gradients in the backward pass at reduced memory cost, without increasing the computational complexity linearly. Our framework allows to use the implementation of the checkpointed gradients, and therefore, besides allowing for bigger batches due to the multi-GPU support, it also allows for bigger per-GPU batches.

#### E. Graph Freezing for Deployment

Once the trained model performs as desired, the framework exports a log directory containing a copy of all the configuration files used, for later reference, and two directories inside containing the best IoU and best accuracy checkpoints.

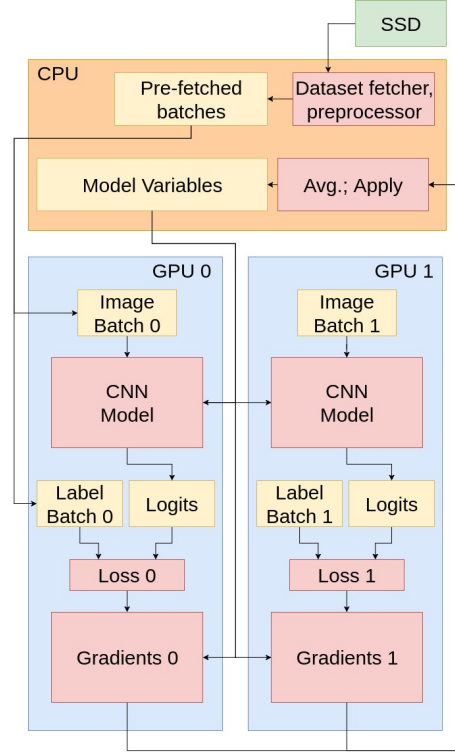


Fig. 4. Multi-GPU training. Example using two GPUs, but scalable to all GPUs available in workstation.

To deploy the model and use it with different back-ends, such as TensorRT, we need to “freeze” the desired model, which means removing all of the helper operations required for training and unnecessary for inference, such as the optimizer ops, the gradients, dropout, and calculation of train-time batch normalization momentums. The abstract network provides a method which handles this procedure, creating another directory with four frozen models: the model in NCHW format, which is faster when inferring using GPUs; the model in NHWC format, which can be faster when using CPUs; an optimized model which tries to further combine redundant operations, and an 8-bit quantized model for faster inference. This method also generates a new configuration file called *nodes.yaml*, which contains important node names, such as the inputs, code, and outputs as logits, softmax, and argmax. This allows for a more automated parsing of the frozen model during inference, automatically remembering the names of the inputs and outputs. We provide a Python script for this procedure, which takes in a log directory and outputs all the frozen models and their configuration files in a packaged directory that contains all files needed for deployment. We also provide other applications to test this model in images and videos, in order to observe the performance qualitatively for debugging, and to serve as an example for serving using python, in case this is desired. It is key to notice that since the whole process can be performed in a host PC, the device PC on the robot only needs the dependencies to run the inference, such as our C++ library.

Listing 1. C++ code showing simplicity of semantic segmentation CNN inference in C++ application, using Bonnet framework as a library.

```
#include <bonnet.hpp>
#include <opencv2/core/core.hpp>
#include <string>

int main() {

    // path to frozen dir
    std::string path = "/path/frozen/pb";
    // tf for Tensorflow, trt for TensorRT
    std::string backend = "trt";
    // gpu or cpu (or specialized)
    std::string dev = "/gpu:0";

    // Create the network
    bonnet::Bonnet net(path, backend, dev);

    // Infer image from disk
    cv::Mat image, mask, mask_color;
    image = cv::imread("/path/to/image");
    net.infer(image, mask);

    // If necessary, colorize (makes
    // color debug mask like Fig.1)
    net.color(mask, mask_color);

}
```

## V. BONNET DEPLOYMENT

For the deployment of the model on a real robot, we provide a C++ library with an abstract handler class that takes care of the inference of semantic segmentation, and allows for each implemented back end to run without changes in the API level. The library can handle inference from a frozen model that is generated through the last step of the Python interface. Bonnet handles the inference through the user’s selection of the desired back end, execution device (GPU, CPU, or other accelerators), and the frozen model to use. There are two ways to access this library: one is by linking it with an existing C++ application, using the two provided standalone applications as a usage example, and the other one is to use the provided ROS node, which already takes care of everything needed to do the inference, from debayering the input images, to resizing, and publishing the mask topics. List. 1 contains an example of how to build a small “main.cpp” application to perform semantic segmentation on an image from disk using our C++ library.

## VI. SAMPLE USE CASES SHIPPED WITH BONNET

In order to show the capabilities of the framework, we provide a sample architecture focusing on a good accuracy vs. runtime trade-off. The model is based on the concept of “variable receptive field non-bottleneck” behind ERFNet [23] which proposes factorized convolutions of diverse receptive fields in order to widen each layer of a residual and asymmetric encoder-decoder architecture (see Fig. 2), to improve the performance of real-time capable models such as ENet, without increasing the computational cost significantly. Tab. I

TABLE I  
PIXEL-WISE METRICS FOR SAMPLE ARCHITECTURE.

Dataset	Input Size	#Params	#Operations	mIoU	mAccuracy
Cityscapes	512x256	1.8M	16.5B	52.3%	90.4%
	1024x512		66.1B	64.1%	93.1%
Synthia	512x384	1.8M	24.2B	64.1%	92.3%
	960x720		85.1B	71.3%	95.2%
Crop-Weed	512x384	1.1M	9.5B	80.1%	98.5%

TABLE II  
MEAN RUNTIME OF THE SAMPLE ARCHITECTURE.

Dataset	Input Size	Back-end	i7+GTX1080Ti	Jetson TX2 SoC
Cityscapes	512x256	TensorFlow	19ms $\approx$ 52 Hz	170ms $\approx$ 6 Hz
		TensorRT	10ms $\approx$ 100 Hz	89ms $\approx$ 11 Hz
	1024x512	TensorFlow	71ms $\approx$ 14 Hz	585ms $\approx$ 2 Hz
		TensorRT	33ms $\approx$ 30 Hz	245ms $\approx$ 4 Hz
Synthia	512x384	TensorFlow	20ms $\approx$ 50 Hz	223ms $\approx$ 4 Hz
		TensorRT	11ms $\approx$ 100 Hz	127ms $\approx$ 8 Hz
	960x720	TensorFlow	61ms $\approx$ 16 Hz	673ms $\approx$ 1 Hz
		TensorRT	27ms $\approx$ 37 Hz	362ms $\approx$ 3 Hz
Crop-Weed	512x384	TensorFlow	9ms $\approx$ 111 Hz	132ms $\approx$ 8 Hz
		TensorRT	4ms $\approx$ 250 Hz	99ms $\approx$ 10 Hz

shows the performance of the sample architecture on three diverse challenging datasets, two for scene parsing, and one for agricultural purposes, for which we provide the trained weights. Because each problem presents a different level of difficulty, and uses images of a different aspect ratio, we show the performance of the model for different number of parameters and number of operations by varying the number of kernels of each layer of the base architecture and the size of the input.

The model size used for the Cityscapes dataset, which contains a set of 5000 challenging real world images for road scene parsing (of which 2975 are used for training) was shared with the Synthia dataset, which contains over 10000 synthetic images for road scene parsing in different lighting and seasonal conditions, and it was pre-trained with the latter in order to increase the performance on the real world data.

The model used for the agricultural dataset is reduced in size, given the lower complexity of the task due to the reduced number of classes, and it can therefore benefit from a faster running time.

Since the framework is meant to serve as a general starting point to implement different architectures, we advise referring to the code in order to have an up-to-date measure of the latest architecture design performances.

As previously stated, our sample model is designed with a focus on the best accuracy vs. runtime trade-off currently possible for robotics. Therefore, these architectures achieve values of accuracy and intersection over union scores which are slightly inferior to-state-of-the-art papers in computer vision, but are specially fast to run, which is critical in real-time applications such as robotics, where processing images at the frame-rate of the camera device is desirable, and often necessary. Tab. II shows the runtime of the included models,

depending on the model and input size, where we can see how much the inference time can be improved by using custom accelerators for the available commercial hardware, further supporting the importance of allowing the user to transparently benefit from its usage.

## VII. CONCLUSION

In this paper, we presented Bonnet, an open-source semantic segmentation training and deployment framework for robotics research. Our framework eases the integration of state-of-the-art semantic segmentation methods in robotics. It provides a stable interface allowing the community to better collaborate, add different datasets and network architectures, and share implementation efforts as well as pre-trained models. We believe that this framework will speed up the deployment of semantic segmentation CNNs on research robotics platforms. We provide a sample architecture that—depending on the available hardware and input image size—operates at camera framerate or even faster, including pre-trained weights for diverse and challenging datasets with the goal that the robotics community will exploit them and contribute to the framework.

## REFERENCES

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] A. Armagan, M. Hirzer, and V. Lepetit. Semantic segmentation for 3d localization in urban environments. In *Joint Urban Remote Sensing Event (JURSE)*, pages 1–4, 2017.
- [3] N. Atanasov, M. Zhu, K. Daniilidis, and G.J. Pappas. Localization from semantic observations via the matrix permanent. *Intl. Journal of Robotics Research (IJRR)*, 35(1-3):73–99, 2016.
- [4] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI)*, 2017.
- [5] N. Blodow, L. C. Goron, Z. C. Marton, D. Pangercic, T. Rühr, M. Tenorth, and M. Beetz. Autonomous semantic mapping for robots performing everyday manipulation tasks in kitchen environments. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 4263–4270, 2011.
- [6] N. Chebrolu, P. Lottes, A. Schaefer, W. Winterhalter, W. Burgard, and C. Stachniss. Agricultural Robot Dataset for Plant Classification, Localization and Mapping on Sugar Beet Fields. *Intl. Journal of Robotics Research (IJRR)*, 2017.
- [7] L.C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A.L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *arXiv preprint*, abs/1606.00915, 2016.
- [8] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint*, abs/1604.06174, 2016.
- [9] M. Cordts, S. Mohamed Omran, Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [10] R. Drouilly, P. Rives, and B. Morisset. Semantic representation for navigation in large-scale environments. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1106–1111, May 2015.
- [11] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint*, abs/1706.02677, 2017.
- [12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [13] G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, abs/1207.0580, 2012.
- [14] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint*, abs/1502.03167, 2015.
- [15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint*, abs/1408.5093, 2014.
- [16] R. Khanna, M. Möller, J. Pfeifer, F. Liebisch, A. Walter, and R. Siegwart. Beyond point clouds - 3d mapping and field parameter measurements using uavs. In *IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, 2015.
- [17] D.P. Kingma and J.Ba. Adam: A method for stochastic optimization. *arXiv preprint*, abs/1412.6980, 2014.
- [18] T.Y. Lin, P. Goyal, R.B. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *arXiv preprint*, abs/1708.02002, 2017.
- [19] P. Lottes, H. Markus, S. Sander, M. Matthias, S.L. Peter, and C. Stachniss. An Effective Classification System for Separating Sugar Beets and Weeds for Precision Farming Applications. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2016.
- [20] A. Milioto, P. Lottes, and C. Stachniss. Real-time semantic segmentation of crop and weed for precision agriculture robots leveraging background knowledge in cnns. *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2018. In Press.
- [21] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello. Enet: Deep neural network architecture for real-time semantic segmentation. *arXiv preprint*, abs/1606.02147, 2016.
- [22] J. Pöschmann, P. Neubert, S. Schubert, and P. Protzel. Synthesized semantic views for mobile robot localization. In *Proc. of the Europ. Conf. on Mobile Robotics (ECMR)*, pages 1–6, 2017.
- [23] E. Romera, J. M. Alvarez, L. M. Bergasa, and R. Arroyo. Erfnet: Efficient residual factorized convnet for real-time semantic segmentation. *IEEE Trans. on Intelligent Transportation Systems (ITS)*, 19(1):263–272, Jan 2018.
- [24] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *arXiv preprint*, abs/1505.04597, 2015.
- [25] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.C. Chen. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *arXiv preprint*, January 2018.
- [27] M. Schwarz, A. Milan, A.S. Periyasamy, and S. Behnke. Rgb-d object detection and semantic segmentation for autonomous manipulation in clutter. *Intl. Journal of Robotics Research (IJRR)*, 2017.
- [28] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint*, abs/1409.1556, 2014.
- [29] N. Sünderhauf, F. Dayoub, S. McMahon, B. Talbot, R. Schulz, P.I. Corke, G. Wyeth, B. Upcroft, and M. Milford. Place categorization and semantic mapping on a mobile robot. *arXiv preprint*, abs/1507.02428, 2015.
- [30] C. Zhao, H. Hu, and D. Gu. Building a grid-point cloud-semantic map based on graph for the navigation of intelligent wheelchair. In *21st International Conference on Automation and Computing (ICAC)*, pages 1–7, Sept 2015.
- [31] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia. Icnnet for real-time semantic segmentation on high-resolution images. *arXiv preprint*, abs/1704.08545, 2017.
- [32] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network. *arXiv preprint*, abs/1612.01105, 2016.
- [33] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. Torr. Conditional random fields as recurrent neural networks. In *Proc. of the IEEE Intl. Conf. on Computer Vision (ICCV)*, 2015.