ML-Caffe-Segmentation-Tutorial

The following is a **tutorial** on how to **train, quantize, compile, and deploy** various segmentation networks including: **ENet, ESPNet, FPN, UNet, and a reduced compute version of UNet that we'll call Unet-lite**. The training dataset used for this tutorial is the **Cityscapes dataset**, and the **Caffe** framework is used for training the models. After training, the **DNNDK tools**are used to quantize and compile the models, and ARM C++ application examples are included for deploying the models on a Xilinx **ZCU102** target board. For background information on ESPNet, ENet, and general segmentation approaches, the Segmentation Introduction Presentation has been provided.

Note that the goal of this tutorial is not to provide optimized high accuracy models, but rather to provide a framework and guidance under which segmentation models can be trained and deployed on Xilinx MPSoCs.

The tutorial is organized as follows:

1) Environment Setup and Installation

2) Prepare the Cityscapes database for Training Segmentation Models

3) Training the Models

4) Evaluating the Floating Point Models on the Host PC

6) Running the Models on the ZCU102

7) Post processing the Hardware Inference Output

Pre-Install Considerations for Caffe for Segmentation Models

Segmentation Networks require a special fork of Caffe. The Caffe distribution needed for these networks is included with this example in the Segment/caffe-master directory. Instructions to install and configure the ZCU102 image and install the provided Caffe distribution directly on the host are available via step "1.0 Environment Setup and Installation".

Whatever solution you have used to setup the environment, from now on I assume that you have the provided Caffe fork installed (either directly on your machine, in a python virtual environment, or within the Docker image) on your Ubuntu 16.04 Linux file system, and you have set the **$CAFFE_ROOT** environment variable with the following command into your **~/.bashrc**file (modify the file path for the location where the Segment directory has been placed):

```
export CAFFE_ROOT=/absolute_path_to_caffe_install/Segment/caffe-master
```

If you have successfully installed the provided fork of Caffe, the DNNDK tools, and configured your ZCU102 board with the Debian stretch image then proceed to "2.0 Prepare the Cityscapes database for Training Segmentation Models", otherwise, proceed to step "1.0 Environment Setup and Installation".

This text from this tutorial has also been converted to a .pdf format and can be found in PDF/tutorial.pdf. Note that the .pdf is not maintained, and only the .md files are maintained, so please use the native git format for the latest.

1.0 Environment Setup and Installation

An Ubuntu 16.04 or 14.04 host machine is needed with an Nvidia GPU card as well as the proper CUDA, CuDNN, and NCCL libraries (these dependencies are documented in the user guide which is included in the subsequent content)

This tutorial was tested with a ZCU102 revision 1.0 (newer should also be ok), Displayport monitor, keyboard, mouse, and USB hub with USB micro converter.

The DNNDK release used for testing this tutorial is xlnx_dnndk_v3.0_190624.tar.gz, MD5=51d0b0fe95493a2e0bf9c19116b17bb8.

Site for downloading DPU/DNNDK images: https://www.xilinx.com/products/design-tools/ai-inference/ai-developer-hub.html

(This image needs to be written to an SD card using Win32DiskImager or equivalent)

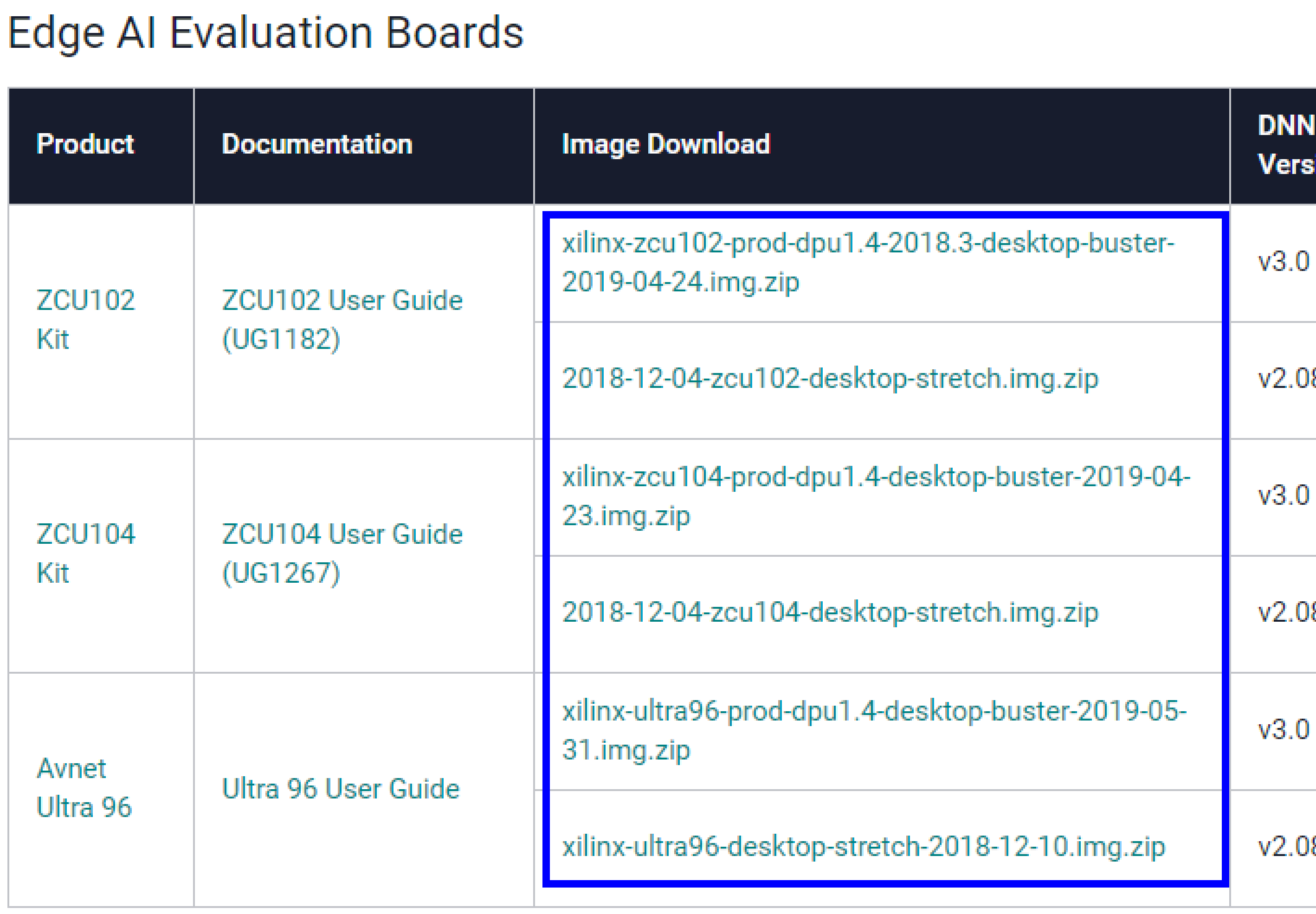
DNNDK user guide Tools for ZCU102 and Host x86: https://www.xilinx.com/support/documentation/user_guides/ug1327-dnndk-user-guide.pdf

1.1 PART 1: Deephi DNNDK and ZCU102 Image Setup:

The Deephi tools and image can be set up via the following steps:

1. Download the demo image for the desired board. The images available on the AI Developer Hub support the Ultra96, ZCU104, and ZCU102. This tutorial should work with any of the boards provided that dnnc is set to target the correct DPU within that board image. For the Ultra96, the DPU target called in the dnnc command line should be 2304FA, and the ZCU102 and ZCU104 should be the 4096FA.

https://www.xilinx.com/products/design-tools/ai-inference/ai-developer-hub.html



Edge AI Evaluation Boards

Product	Documentation	Image Download	DNN Vers
ZCU102 Kit	ZCU102 User Guide (UG1182)	xilinx-zcu102-prod-dpu1.4-2018.3-desktop-buster-2019-04-24.img.zip	v3.0
		2018-12-04-zcu102-desktop-stretch.img.zip	v2.08
ZCU104 Kit	ZCU104 User Guide (UG1267)	xilinx-zcu104-prod-dpu1.4-desktop-buster-2019-04-23.img.zip	v3.0
		2018-12-04-zcu104-desktop-stretch.img.zip	v2.08
Avnet Ultra 96	Ultra 96 User Guide	xilinx-ultra96-prod-dpu1.4-desktop-buster-2019-05-31.img.zip	v3.0
		xilinx-ultra96-desktop-stretch-2018-12-10.img.zip	v2.08

Note that all networks have been tested with DNNDK v3.0 and the DPU v1.4.0 and have been found to work properly with this configuration.

2. Extract the iso image from the zip file

3. If using Windows, install Win32DiskImager and flash the image to a 16GB or larger SD card (this tutorial was verified using a SanDisk Ultra 80MB/s 32GB SD card)

4. Boot the ZCU102 from the SD card with a USB keyboard and monitor connected. If unsure how to configure the ZCU102 jumpers and switches, please refer to the reVISION Getting Started Guide Operating Instructions

   4.a) If for some reason no display is shown on the monitor, it is recommended to try executing the following commands via the serial terminal:

   ```
   export DISPLAY=:0.0
   xrandr --output DP-1 --mode 1024x768
   xset -dpms
   ```

5. If using Windows, WinSCP, PSCP, or MobaXTerm can be used to transfer the files between the ZCU102 and the host PC. If using PSCP, you can simply copy PSCP from ref_files/ to the desired directory, cd to that directory with the Windows command prompt and use it. If using WinSCP or MobaXTerm, you can connect to the board with the GUI and transfer the files by dragging and dropping them.

6. Next connect an Ethernet cable from the ZCU102 board to your host machine and get/set the Ethernet IP address of the board using ifconfig in a terminal (I set mine to 192.168.1.102 using `ifconfig eth0 192.168.1.102`).

7. Set the IP of your host machine to static on the same subnet such as 192.168.1.101. If there is an anti-virus firewall on your host machine, you may need to disable it before proceeding to the next step.

8. The DNNDK tools and sample images need to be copied to the board. This can be done with pscp using a command like the following (password is root): `pscp –r c:\pathtodnndkfiles root@192.168.0.102:/root/` or by dragging and dropping with WinSCP or or MobaXTerm.

9. Copy over the ZCU102 folder and the common folder included in the DNNDK tools and sample images downloaded from step 9 from the host machine to the ZCU102 board.

10. Install the DNNDK tools on the ZCU102 by running the provided install.sh script (./ZCU102/install.sh).

11. The x86 host tools will need to be copied to a linux x86 host machine to quantize/compile the model. This step will be covered in the subsequent steps.

11.a) NOTE: To change the display resolution in the Ubuntu linux, execute the following:

`xrandr – q` to list the supported modes
`xrandr -s 1920x1080` to set the mode

1.1 PART 2: Installing the Host Dependencies

The following guidelines are provided for direct setup on a host x86 Ubuntu machine with GPU:

1. Install the dependencies:

```
sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev
libboost-all-dev libhdf5-serial-dev python-numpy python-scipy python-matplotlib
python-sklearn python-skimage python-h5py python-protobuf python-leveldb python-
networkx python-nose python-pandas python-gflags ipython protobuf-c-compiler
protobuf-compiler libboost-regex-dev libyaml-cpp-dev g++ git make build-essential
autoconf libtool libopenblas-dev libgflags-dev libgoogle-glog-dev liblmdb-dev
libhdf5-dev libboost-system-dev libboost-thread-dev libboost-filesystem-dev python-
opencv libyaml-dev
```

2. Install the NVidia libraries/drivers -This tutorial was verified with the following configuration:

NVidia GTX 1080ti Graphics Card

NVidia 390 graphics drivers

2.a) CUDA v8.0 (follow Nvidia instructions and install from the runfile along with the patch)

2.b) CuDNN v7.0.5 using the "cuDNN v7.0.5 Library for Linux" selection for CUDA 8.0 The following steps were used to install CuDNN:

```
  sudo tar -xzvf cudnn-9.1-linux-x64-v7.tgz
  sudo cp cuda/include/cudnn.h /usr/local/cuda/include
  sudo ln –s cuda/lib64/libcudnn* /usr/local/cuda/lib64
  sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

2.c) NCCL v1.2.3 - The following steps were used to install:

Download NCCL 1.2.3 (linked with CUDA 8.0) source code (tar.gz)

```
   tar –xvf nccl-1.2.3-1-cuda8.0.tar.gz
   cd nccl-1.2.3-1-cuda8.0
   sudo make install -j
   sudo ldconfig /usr/local/cuda/lib64
```
2.d) Next create symbolic links and an environment variable for hdf5:

```
   cd /usr/lib/x86_64-linux-gnu
   sudo ln –s libhdf5-serial.so.x.x.x libhdf5.so
   sudo ln –s libhdf5_serial_hl.so.x.x.x libhdf5_hl.so
   export CPATH="/usr/include/hdf5/serial"
```
This last line can also be added to ~/.bashrc to configure it on startup

2.e) reboot your machine

NOTE: If issues are encountered where the graphics driver needs to be installed, please use the following instructions to install:

a. First remove other installations via the following:

```
  sudo apt-get purge nvidia-cuda*
  sudo apt-get purge nvidia-*
```
b. Enter a terminal session using ctrl+alt+F2

c. Stop lightdm: sudo service lightdm stop

d. Create a file at /etc/modprobe.d/blacklist-nouveau.conf with the following contents:

blacklist nouveau

options nouveau modeset=0

e. Then do: `sudo update-initramfs –u`
f. Add the graphics driver PPA:

```
  sudo add-apt-repository ppa:graphics-drivers
  sudo apt-get update
```
g. Now install and activate the latest drivers (for this tutorial, version 390):

`sudo apt-get install nvidia-390`

1.1 PART 3: Installing Caffe Fork for Segmentation

1. Copy the provided distribution of Caffe located at in the Segment folder to your host Ubuntu machine. Make sure to copy the entire Segment folder, not just the "caffe-master" subfolder. The reason is that all of these folders will be needed later in the tutorial. Specifically, the "workspace" directory contains the prototxt descriptions needed for training the various models as well as scripts for

evaluating the models, and the "DNNDK" folder contains the workspace for quantizing/compiling the models for execution on the ZCU102, while the "caffe-master" folder contains the caffe fork needed to train these networks.

2. Note that the Makefile.Config (as well as the Makefile.Config.example) has already been modified to enable the use of CuDNN and has set the CUDA_DIR to /usr/local/cuda. I have already modified this file, so you should not need to do anything to it to build the caffe distribution. Note that the LIBRARY_DIRS and INCLUDE_DIRS variables have also been set in this file to point to the following locations for hdf5:

```
  INCLUDE_DIRS:=$(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/serial

  LIBRARY_DIRS:=$(PYTHON_LIB) /usr/local/lib /usr/lib /usr/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu/hdf5/serial
```

3. At this point, you should be able to cd to your caffe-master directory (this is the directory where the makefile exists – hereafter referred to as $CAFFE_ROOT) and run the following:

```
  make clean
  make –j8
  make py
```

6. Next run the following command to export the PYTHONPATH variable (note that if you change terminal windows, you will need to re-export this variable):

```
export PYTHONPATH= $CAFFE_ROOT/python:$PYTHONPATH
```

At this point, Caffe is now installed and you may proceed to the next section on dataset preparation.

2.0 Prepare the Cityscapes database for Training Segmentation Models

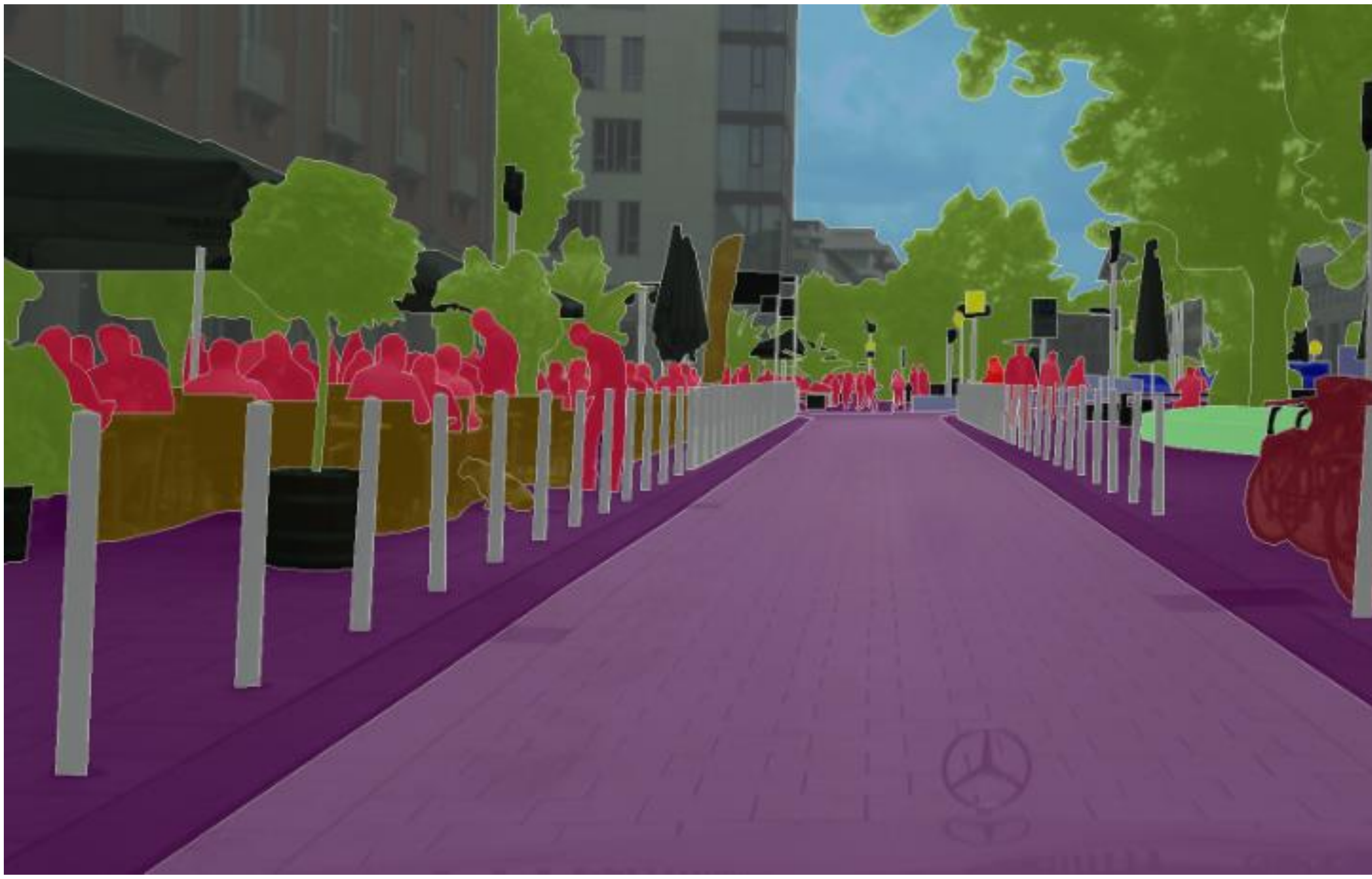
For this tutorial, we’ll be training the models on the Cityscapes dataset. Cityscapes is an automotive dataset created by Daimler which includes various driving scenes, mostly contained in Germany.

The files from Cityscapes provide around 5000 images with fine annotations (class labels) for various city driving scenarios. There are two primary folders from the dataset that we'll be working with:

- leftImg8bit (includes all of the input images for training)
- gtFine (includes the class annotations as polygonal format (.json files))

There are also scripts that are separately downloaded for the dataset which are used to transform the class label .json files into class label images (.png files) which are used for training.

The following is an example which shows the different classes after being color coded and alpha-blended with the original image.



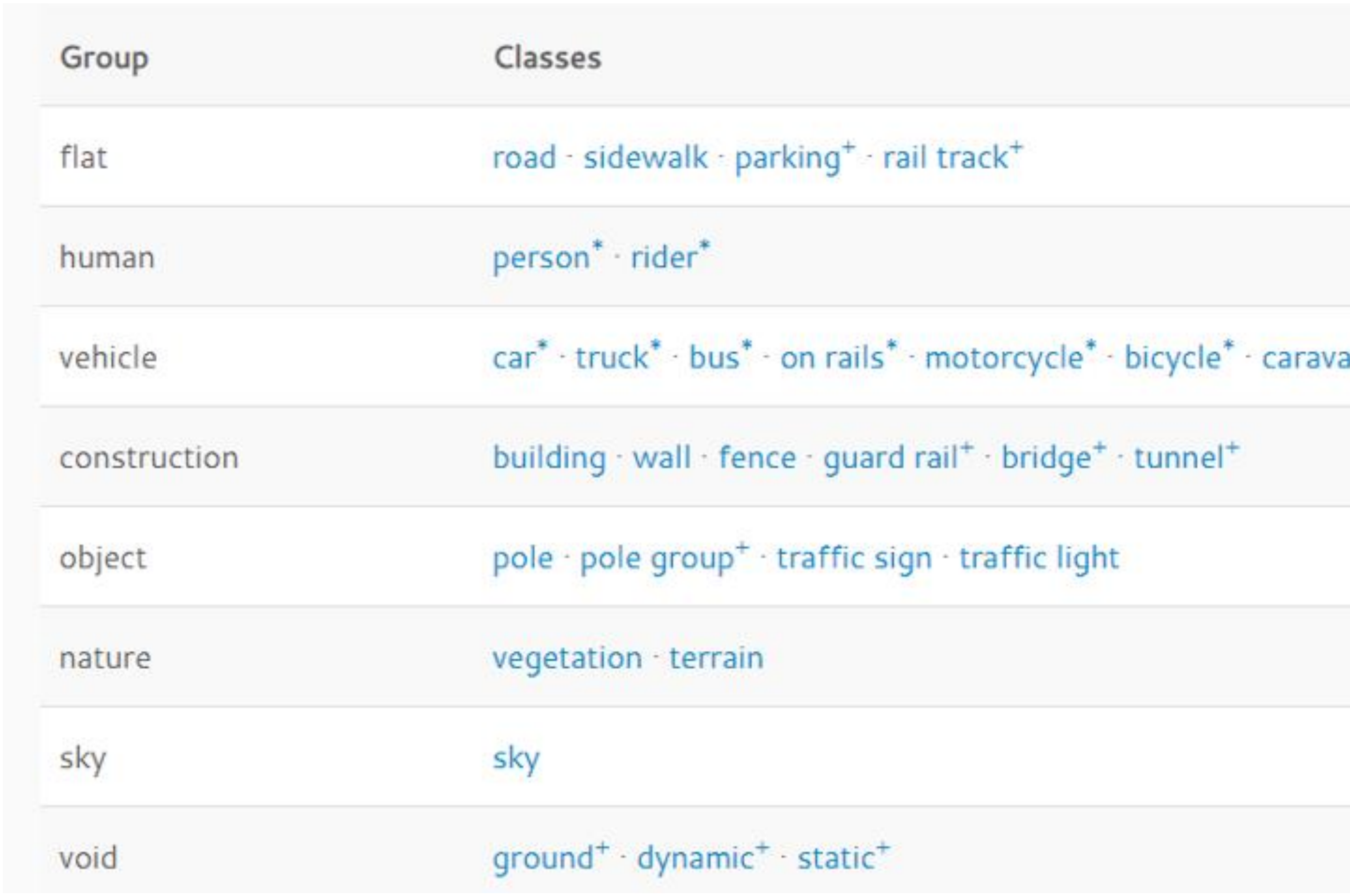
Citation: https://www.cityscapes-dataset.com/examples/

The focus of this database for our purpose is on Semantic Annotations which consist of the following types (we only use the Fine Annotations for this tutorial, though it should also be possible to use coarse annotations and perhaps achieve even better results):

Course Annotations (20000 images)

Fine Annotations (5000 images)

There are 8 groups contained within the Cityscapes dataset with 19 classes. In the following figure, it can be seen that there are 30 classes listed, but all classes with a ‘+’ next to them are treated as a single void class and preparation steps will change their values to ‘255’ which will subsequently be ignored in the training process:



Group	Classes
flat	road · sidewalk · parking^{+} · rail track^{+}
human	person* · rider*
vehicle	car* · truck* · bus* · on rails* · motorcycle* · bicycle* · carava
construction	building · wall · fence · guard rail^{+} · bridge^{+} · tunnel^{+}
object	pole · pole group^{+} · traffic sign · traffic light
nature	vegetation · terrain
sky	sky
void	ground^{+} · dynamic^{+} · static^{+}

Citation: https://www.cityscapes-dataset.com/dataset-overview/

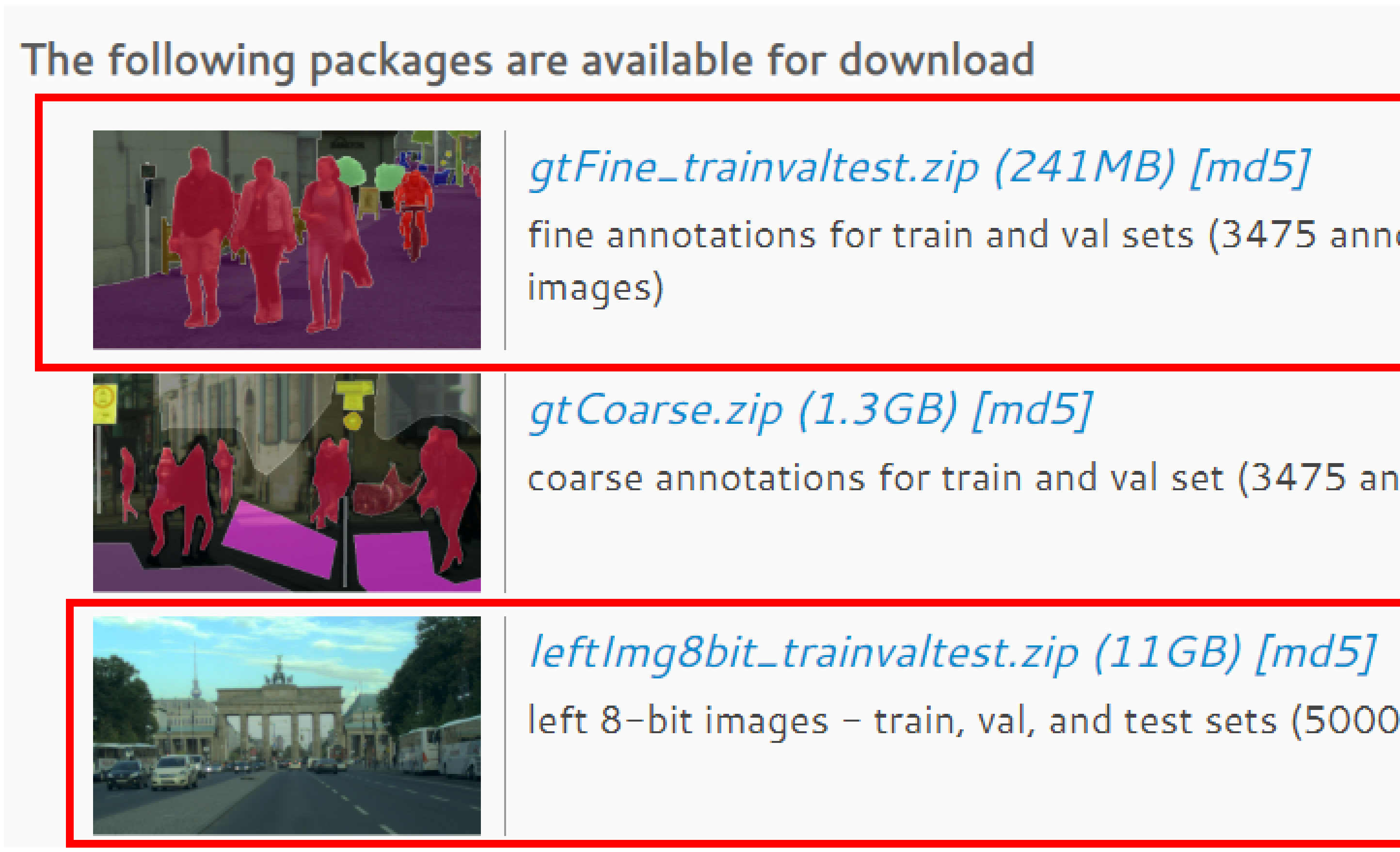
More information about the database can be found at the following URL:

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

Since preparing such a database for training requires a number of steps, the following detailed instructions are provided:

1. Download the Cityscapes dataset from: https://www.cityscapes-dataset.com/downloads/

The specific packages needed are the gtFine_trainvaltest.zip and lefImg8bit_trainvaltest.zip as shown in the figure below. These files include the 5000 images with fine (pixel-wise) semantic annotations which are divided into train, test, and validation groups. It would also be possible to train using the coarse annotations provided by Cityscapes and perhaps achieve better results, but only training with the fine annotations is covered in this tutorial.


The following packages are available for download

gtFine_trainvaltest.zip (241MB) [md5]
fine annotations for train and val sets (3475 ann
images)

gtCoarse.zip (1.3GB) [md5]
coarse annotations for train and val set (3475 an

leftImg8bit_trainvaltest.zip (11GB) [md5]
left 8-bit images - train, val, and test sets (5000

Citation: https://www.cityscapes-dataset.com/downloads/

2. Extract these files into a folder on the Linux workstation. After this you should have a folder containing sub-folders labeled "gtFine" and "leftImg8bit". From the introduction, it was noted that these folders contain the class labels and input images.

3. There are various preparation, inspection, and evaluation scripts provided for Cityscapes which can be cloned from github. The next step will be to download or clone these using the following:

```
git clone https://github.com/mcordts/cityscapesScripts.git
```

4. The scripts can be installed by changing directory into the cityscapesScripts folder then using pip:

```
sudo pip install .
```

5. Next we need to export the CITYSCAPES_DATASET variable to point to the directory where you extracted the lefimg8bit and gtFine folders. This environment variable will be used by the preparatory scripts which pre-process the annotations into class labels. In order to do this, first change directory to the location where the dataset was extracted, then run the following command. Consider copy/pasting the command as it uses a backtick character surrounding "pwd" and not a single apostrophe.

```
export CITYSCAPES_DATASET=`pwd`
```

6. The next step is to create the images which have class labels associated with each pixel and set the unused classes to value '255'. This can be done by running the createTrainIdLabelImags.py script. To run this script change directory to the cityscapesScripts/cityscapesscripts directory and run the following:

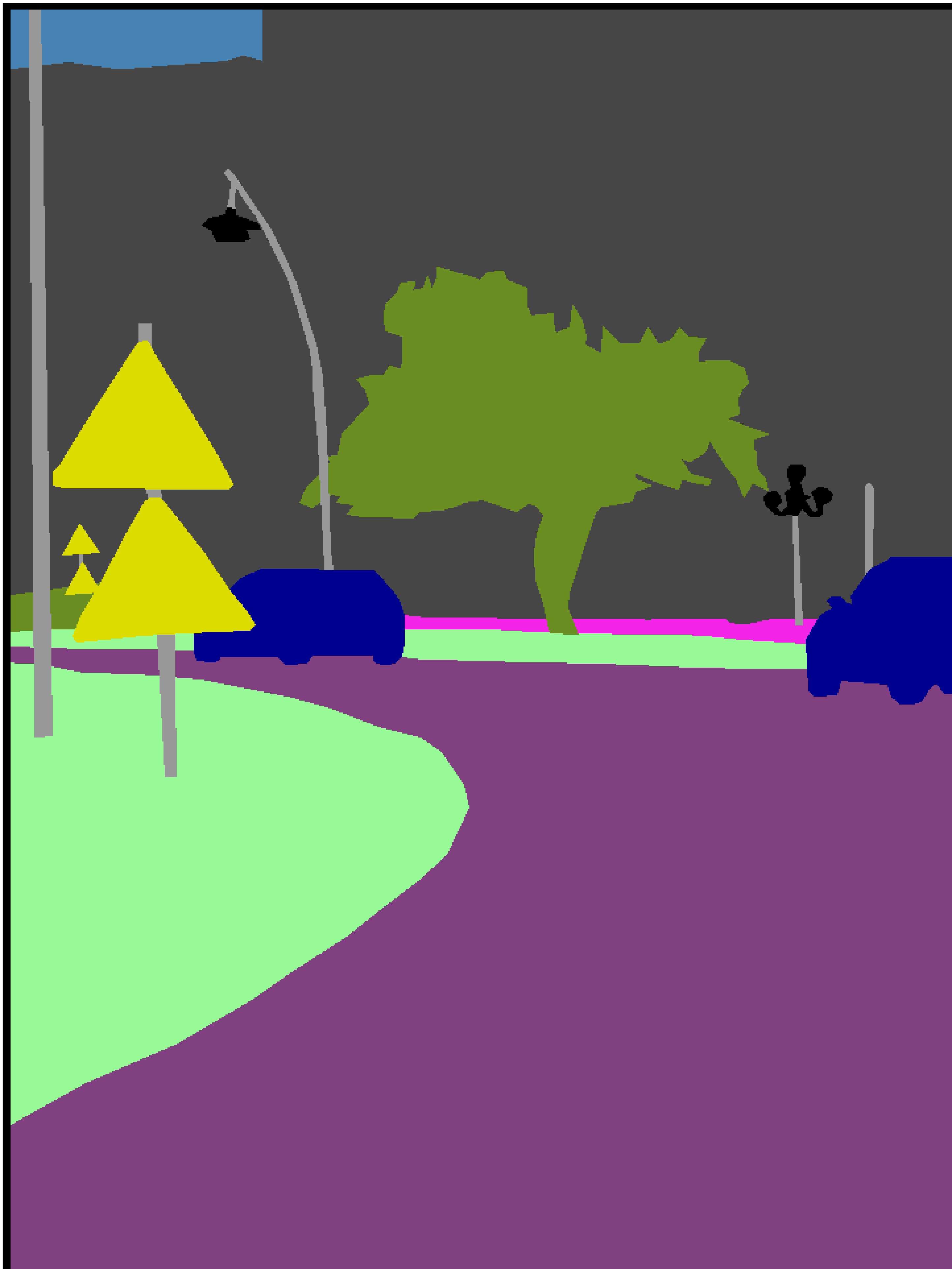
```
python preparation/createTrainIdLabelImags.py
```

This will convert annotations in polygonal format (.json files) to .png images with label IDs, where pixels encode the “train IDs”. Since we use the default 19 classes, you do not need to change anything in labels.py script at this time. We will later go back and change the labels.py for use with evaluating the trained models.

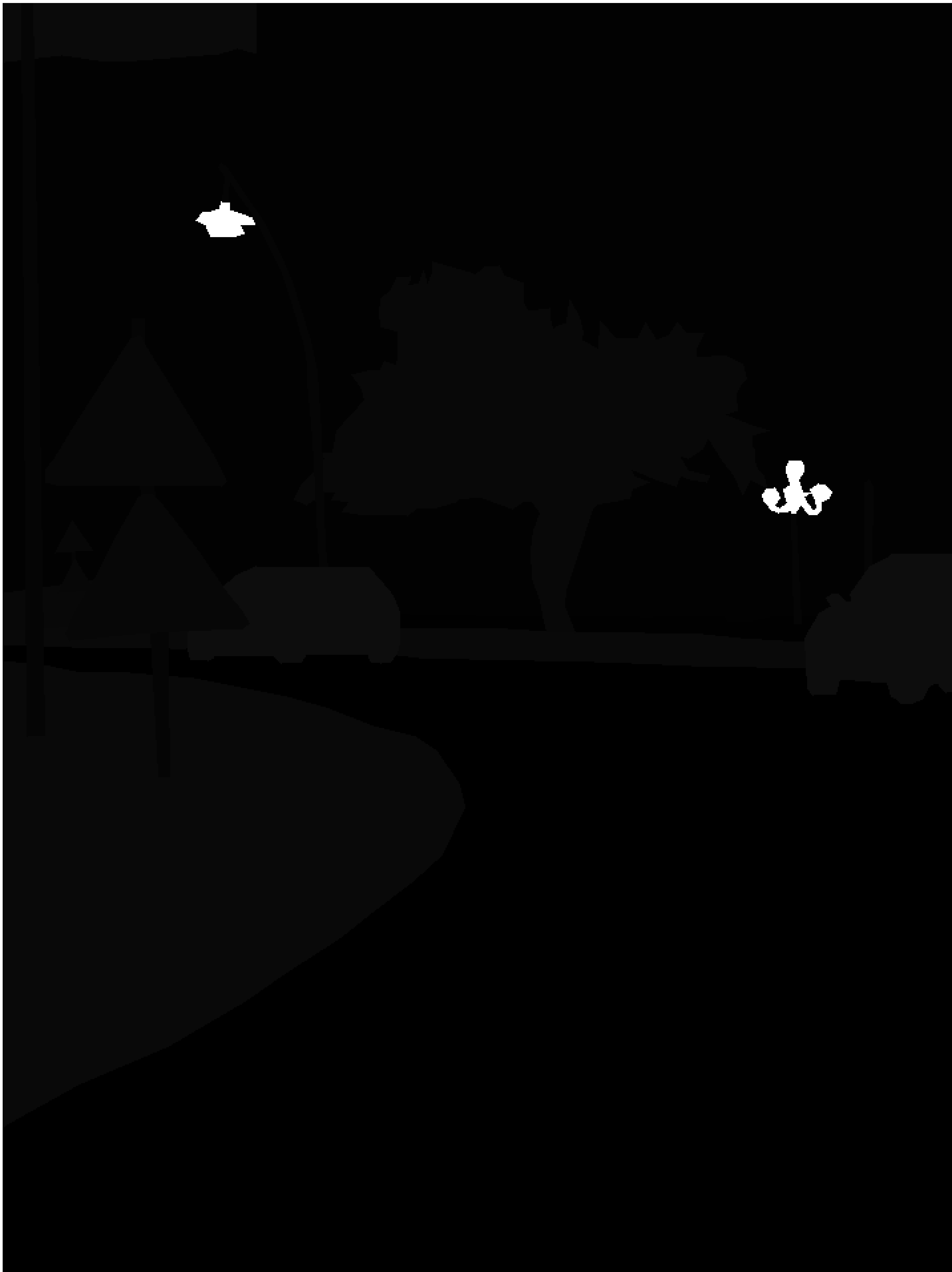
If you are new to datasets, it may be worthwhile to inspect the .json file to see how the polygon data is stored. You'll notice it is basically an array of points that connect the lines for each polygonal area for each class.

After running this script, you will see color coded images which denote the classes as well as trainable images which have the classes encoded in the order determined by the cityscapesscripts/helpers/labels.py.

An example of the color coded image from the dataset is shown here:



Once the pixels are encoded with trainable values, the different classes are identified as values 0-18 and all of the ignored classes are set to value 255. Notice that it's very difficult to distinguish the various classes in this image as they are such low values (with the exception of the ignored classes). An example of the trainable image is shown here:



Note that it is possible to modify the `cityscapesscripts/helpers/labels.py` to change the class annotations during the preparatory step.

7. During the training process, a text file is used (the path to this file is specified in the input data layer of the model prototxt files) to identify the location of the training data and annotations. This text file is located under Segment/workspace/data/cityscapes/img_seg.txt. You will need to modify this text file to point to the absolute paths for the input images and associated label images which were just created, which should exist in the subdirectories where the Cityscapes data was extracted.

For this modification, I use VsCode which is a nice text editor that allows for the use of `alt+shift` to select columns of text and replace them. You can also select a section of text and right click to replace all occurrences of that text.
The left column in the img_seg.txt should point to the input image (these are stored under the `Cityscapes/leftImg8bit`directory), the right column should point to the labelTrainIds.png (which are the annotations or ground truths and are stored under the `Cityscapes/gtFine` directory).
There are many classes that get ignored and their pixel values are set to '255'. You can note that in the provided model prototxt files, the final softmax and accuracy layers in the network have set a label ignore parameter for value 255 to ignore these classes. All of the other classes need to start at class 0 and increment. The prototxt files referred to from here exist in Segment/workspace/model which includes folders for each of the models that are covered in the tutorial.

At this point, the training dataset has been prepared and is ready for use to train the models and you can proceed to the next step which is 3.0 Training Models.

3.0 Training Models

Prototxt files are included which can be used to train the various models. Note that these models may differ somewhat from the original models as they have been modified for end use in the DPU IP. Some of the types of modifications that were made to these models include:

- Replacing the un-pooling layer with deconvolution layer in the decoder module
- Replacing all PReLU with ReLU
- Removing spatial dropout layers
- Replace Batchnorm layers with a merged Batchnorm + Scale layer

- Position Batchnorm layers in parallel with ReLU
- In UNet-full/Unet-lite models Batchnorm/scale layer combinations were inserted before relu layers (after d0c, d1c, d2c, and d3c) as the DPU doesn't support the data flow from Convolution to both the Concat and relu simultaneously

If further analysis is desired, the model prototxt files have been included so they can simply be diff'd from the original caffe prototxt file.

In terms of augmentation, the mean values from the dataset and a scale factor of 0.022 are applied to the input layers for each model.

3.0.1 Training the Models from Scratch

When training from scratch, it is necessary to train ESPNet and ENet models in two stages: For ESPNet, we will train a model similar to the **(c) ESPNet-C** architecture which is shown in the figure below:

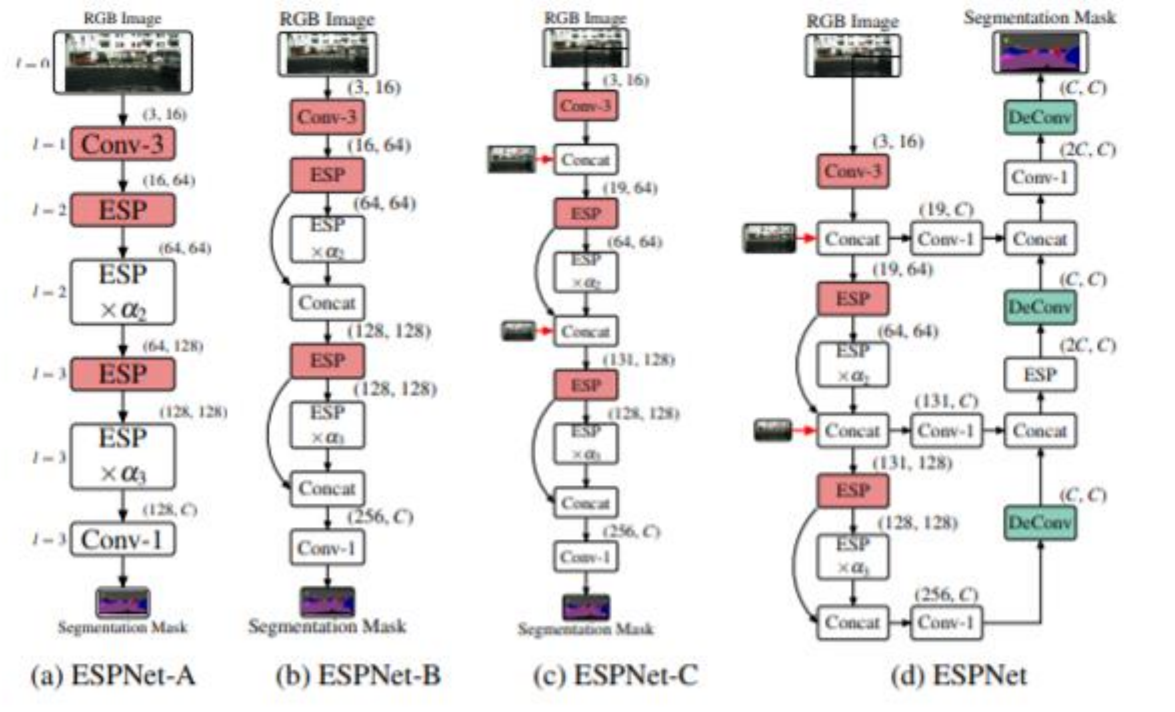


Fig. 4: The path from ESPNet-A to ESPNet. Red and green color boxes represent the modules responsible for down-sampling and up-sampling operations, respectively. Spatial-level l is indicated on the left of every module in (a). We denote each module as (# input channels, # output channels). Here, Conv-n represents $n \times n$ convolution.

Citation: https://arxiv.org/abs/1803.06815

This essentially removes the decoder stage that is present in the **(d) ESPNet** model, and in place of that decoder stage, a single deconvolution layer is added to resize up 8x back to the original input size which matches the annotation size.

For ENet, a similar approach is taken and we train only the encoder stage by removing the decoder portion of the model and adding a single deconvolution layer to resize by a factor of 8x up to the original label size which matches the annotation size.

The FPN, Unet, and Unet-lite models can all be trained end to end, so the encoder/decoder two stage training process is not necessary for those models (though a similar process could be employed if desired and it may end up producing better results).

The pre-trained ESPNet/ENet encoder models were trained for 20K Iterations with an effective batch size of 50, and lr_base 0.0005. Note that larger batch sizes can also be used and may ultimately produce more accurate results, though training time would be increased for larger batch sizes.

If you happen to encounter a situation where you receive a "CUDA out of memory" error, try reducing the batch size in the train_val.prototxt or train_val_encoder.prototxt and increase the corresponding iter_size in the solver.prototxt to maintain the same effective batch size. Note again that all these models were trained with GTX 1080ti GPUs which have 12GB of memory. If your GPU has less memory, you may need to adjust the batch sizes to fit within your GPU memory limits.

After this first step has been completed, we can train the full ESPNet prototxt using the weights from the first step to fine tune the model.

Note that the initial training step for these models takes about 36 hours on my Xeon workstation using a GTX 1080ti graphics card.

3.0.1.1 Training the ESPNet (ESPNet-C) and ENet Encoder Models

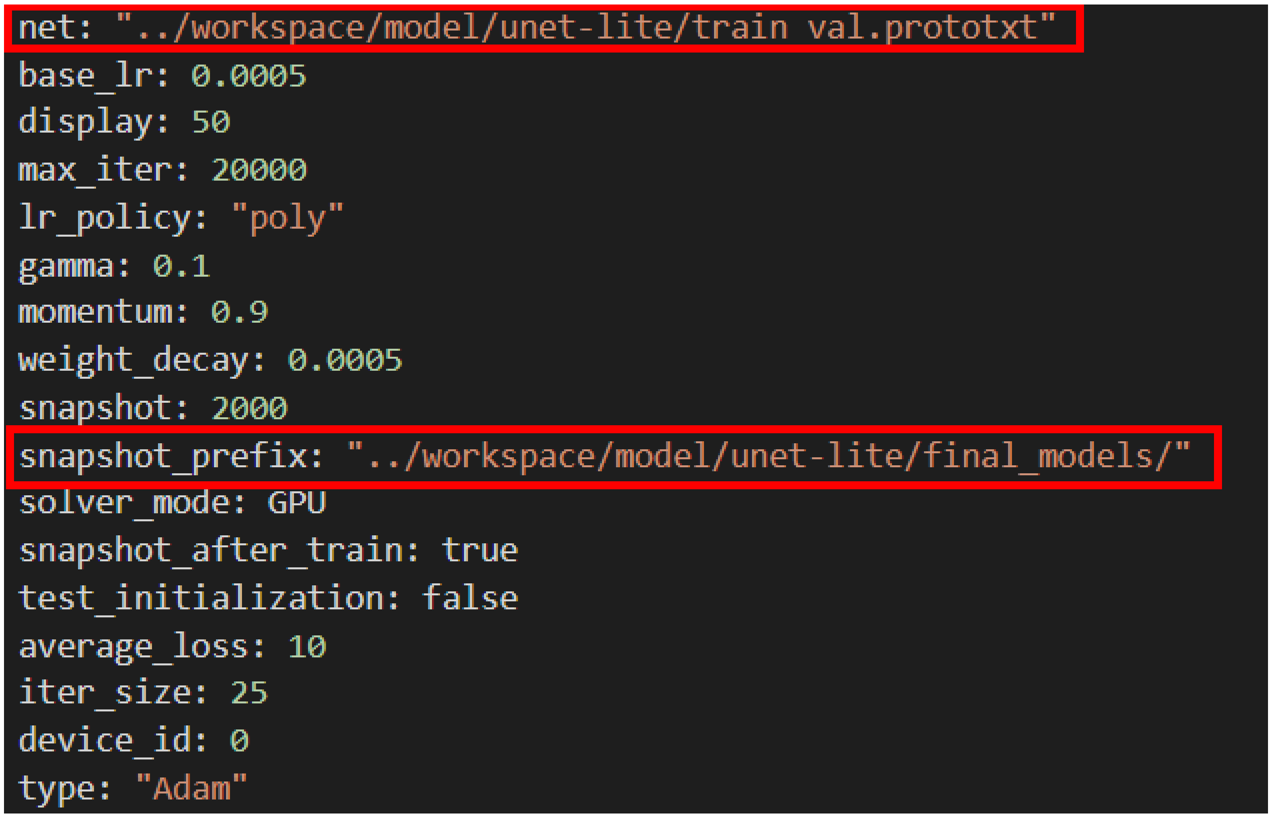
The encoder models have been included with the caffe distribution. The files which will be needed as a starting point for this are the **solver_encoder.prototxt** and the **train_val_encoder.prototxt**. These files are located under the Segment/workspace/model/espnet and Segment/workspace/model/enet paths respectively, and can be used for training the encoder only portions of these networks.

The full model prototxt files are also available under these paths and it is recommended to compare the two files using a text editor to understand what has been removed from the full model for the encoder only portion.

If you would like to skip training the encoder portion of the model, I have included a pre-trained encoder model for both networks which are stored under the Segment/workspace/model/espnet/encoder_models or Segment/workspace/model/enet/encoder_models directories.

1. The first step to train these models is to open the **solver_encoder.prototxt** file for the associated model. It is important to understand the training parameters and paths for this file. Notice the lines containing the "net: " definition and "snapshot_prefix: ".

The first line specifies a relative path to where the train_val_encoder.prototxt exists and the second should point to an existing directory where you would like the model snapshots to be stored. Note that relative paths have been specified so that if run from $CAFFE_ROOT, then no modifications should be needed to run the training.



```
net: "../workspace/model/unet-lite/train_val.prototxt"
base_lr: 0.0005
display: 50
max_iter: 20000
lr_policy: "poly"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
snapshot: 2000
snapshot_prefix: "../workspace/model/unet-lite/final_models/"
solver_mode: GPU
snapshot_after_train: true
test_initialization: false
average_loss: 10
iter_size: 25
device_id: 0
type: "Adam"
```

Note also how the other hyper-parameters are set in the solver prototxt. **The base_lr**, **max_iter**, **iter_size**, and **device_id** are all important training parameters.

The **base_lr** is probably the most important parameter and if it is set to big or too small, the training process will never converge. For this tutorial, it has been found that a size of 0.0005 is an appropriate value for training the models.

The **iter_size** is used to determine the effective batch size. If the batch size is set in the train_val_encoder.prototxt file to '5' in the input layer, then the iter_size essentially applies a multiplier to that batch size by not updating the weights until iter_size number of batches have been completed. For example, if the iter_size is set to 10, then 10 x 5 = 50 is the effective batch size. Batch size has a significant effect on the convergence of the training process as well as the accuracy of the model, so it is important to maintain a larger batch size when training the full models. In the case of this tutorial, this parameter is used to enable the training process to maintain a larger effective batch size where there is a limited amount of GPU memory.

The **device_id** parameter specifies the device id of the GPU card which will be used to accelerate the training process. If you have only one GPU, specify '0', however, multiple GPUs can also be used by using a comma separated list and you can also train multiple models on different GPUs.

As noted before the **max_iter** parameter determines how many times the model will see the training data during the training process. If a dataset has N images and batch size is B, and P is the number of epochs, then the relationship between epochs and iterations is defined as:

Iterations = (N * P) / B

Since the training dataset is around 3000 images, we can re-arrange this equation to calculate the number of epochs by:

(20K*50) /3K = 333 epochs.

2. Once the solver_encoder.prototxt has been verified, the model can be trained by changing directory to $CAFFE_ROOT and running one of the following commands:

For ESPNet:

```
./build/tools/caffe train \
–solver ../workspace/model/espnet/solver_encoder.prototxt \
2>&1 | tee ../workspace/model/espnet/encoder_models/train_encoder_log.txt
```

For ENet:

```
./build/tools/caffe train –solver \
../workspace/model/enet/solver_encoder.prototxt \
2>&1 | tee ../workspace/model/enet/encoder_models/train_encoder_log.txt
```

If errors occur regarding missing libcudart or similar, run `sudo ldconfig /usr/local/cuda/lib64`, then retry.

I have included an example log file from my console output during the training process for ESPNet which is stored under Segment/workspace/model/ESPNet/encoder_models/train_encoder_log.txt. You should see something similar during the training process.

Once the training process has completed, the full model can be trained which uses these pre-trained weights as a starting point for the encoder portion of the model.

3.0.1.2 Training the Full Models

The full models for ENet, ESPNet, FPN, Unet-Full, and Unet-Lite have been included with the caffe distribution. The files which will be needed as a starting point for this are the **solver.prototxt** and the **train_val.prototxt**. These files are located under the Segment/workspace/model/espnet, Segment/workspace/model/enet, Segment/workspace/model/FPN, Segment/workspace/model/unet-full, and Segment/workspace/model/unet-lite paths respectively, and can be used for training the full networks.

Since FPN, Unet-full, and Unet-lite can be trained end-to-end from scratch, there is no need to train the encoder portion separately. Generally for training the full models, a larger batch size is desirable as it helps the model to approximate the full dataset better than a smaller batch size. For this tutorial, I have used batch sizes >= 100 for training the full models.

1. Just like with the encoder training, the first step to train the full models is to open the associated **solver.prototxt** file and view the properties of the various hyper-parameters. Note again that relative paths are used for the "net" and "snapshot_prefix" parameters, so if the training is run from $CAFFE_ROOT, the intended locations should be used for these files.

2. Once the solver.prototxt has been verified, the model can be trained by changing directory to $CAFFE_ROOT and running one of the following commands (assuming the pretrained models are used, otherwise specify the name of your caffemodel):

For ESPNet:

```
./build/tools/caffe train \
-solver ../workspace/model/espnet/solver.prototxt \
-weights ../workspace/model/espnet/encoder_models/pretrained_encoder.caffemodel \
2>&1 | tee ../workspace/model/espnet/final_models/train_log.txt
```

For ENet:

```
./build/tools/caffe train \
-solver ../workspace/model/enet/solver.prototxt \
-weights ../workspace/model/enet/encoder_models/pretrained_encoder.caffemodel \
2>&1 | tee ../workspace/model/enet/final_models/train_log.txt
```

For FPN:

```
./build/tools/caffe train \
-solver ../workspace/model/FPN/solver.prototxt  \
2>&1 | tee ../workspace/model/FPN/final_models/train_log.txt
```

For Unet-Full:

```
./build/tools/caffe train \
-solver ../workspace/model/unet-full/solver.prototxt  \
2>&1 | tee ../workspace/model/unet-full/final_models/train_log.txt
```

For Unet-Lite:

```
./build/tools/caffe train \
-solver ../workspace/model/unet-lite/solver.prototxt \
2>&1 | tee ../workspace/model/unet-lite/final_models/train_log.txt
```

If errors occur regarding missing libcudart or similar, run `sudo ldconfig /usr/local/cuda/lib64`, then retry.

I have included log files for each of the networks which show the output of the training process:

- ENet example log file
- ESPNet example log file
- FPN example log file
- Unet-Full example log file
- Unet-Lite example log file

Note that these are stored respectively at:

```
Segment/workspace/model/enet/final_models/train_log.txt
Segment/workspace/model/espnet/final_models/train_log.txt
Segment/workspace/model/FPN/final_models/train_log.txt
Segment/workspace/model/unet-full/final_models/train_log.txt
Segment/workspace/model/unet-lite/final_models/train_log.txt
```

You should see something similar during the training process for your full models.

In general, training the full models is quite time consuming, in many cases >72 hours per model using my ML workstation.

3.0.2 Training the Models using Transfer Learning

If you would like to accelerate the process of training the models, you can also train from transfer learning using the existing models that I have provided.

The pre-trained full models exist at the following paths:

For ESPNet:

Segment/workspace/model/espnet/final_models/pretrained.caffemodel

For ENet:

Segment/workspace/model/enet/final_models/pretrained.caffemodel

For FPN:

Segment/workspace/model/FPN/final_models/pretrained.caffemodel

For UNet-Full:

Segment/workspace/model/unet-full/final_models/pretrained.caffemodel

For UNet-Lite:

Segment/workspace/model/unet-lite/final_models/pretrained.caffemodel

The following steps can be used to either use transfer learning to retrain only the encoder portion or the full model. The caffemodel that is passed as an argument to the training step should be the appropriate model depending on what the desired approach is:

- If you intend to use transfer learning with the encoder portion only, then use the pre-trained model under the encoder_models directory for the associated network. After this step, you can train the full model using the output of this step as the input weights to train the full model.

- If you intend to use transfer learning with the full model, then use the pre-trained model under the final_models directory for the associated network.

1. Just like with the training from scratch steps, the first step to train the model is to open the associated **solver.prototxt**file and verify the hyper-parameters.

2. Once the solver.prototxt has been verified, the models can be trained by changing directory to $CAFFE_ROOT and running one of the following

commands (modify the weights argument to specify the desired caffemodel for this step):

For ESPNet:

```
./build/tools/caffe train \
–solver ../workspace/model/espnet/solver.prototxt \
–weights ../workspace/model/espnet/final_models/pretrained.caffemodel \
2>&1 | tee ../workspace/model/espnet/caffe-fine-tune-full.log
```

For ENet:

```
./build/tools/caffe train \
–solver ../workspace/model/enet/solver.prototxt \
–weights ../workspace/model/enet/final_models/pretrained.caffemodel \
2>&1 | tee ../workspace/model/enet/caffe-fine-tune-full.log
```

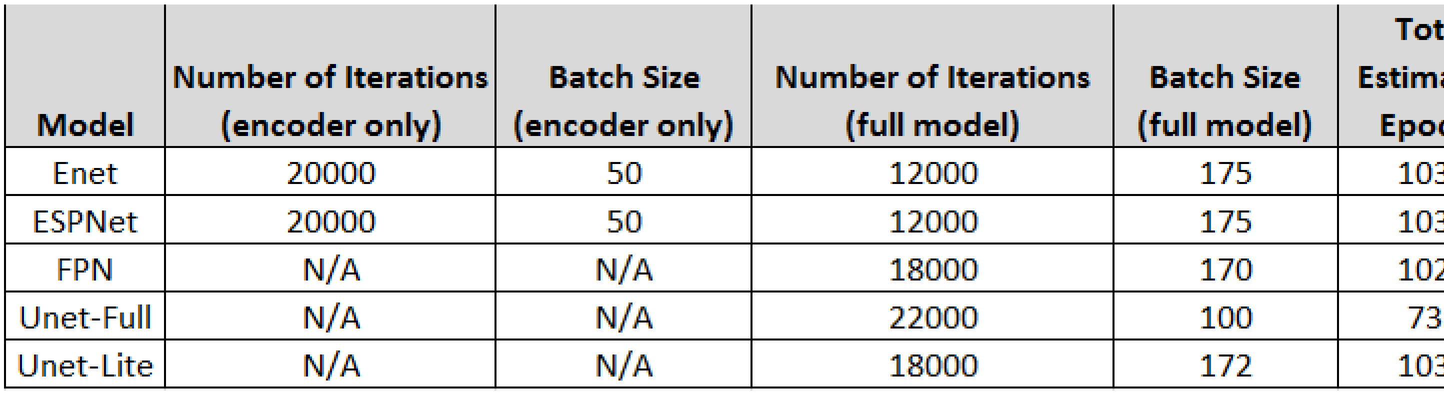
The equivalent commands can be used to perform transfer learning on the other models as well.

If errors occur regarding missing libcudart or similar, run `sudo ldconfig /usr/local/cuda/lib64`, then retry.
At this point, the model training has been completed and you can proceed to the next step which to is evaluate the floating point model, however, if you are interested about the performance/training of the pre-trained models, please take a gander at the next section: "3.1.0 About the Pre-Trained Models".

3.1.0 About the Pre-Trained Models

The pre-trained models included with this tutorial have been trained for various # of iterations and with various batch sizes. Note that training all of the models end to end took about 3-4 weeks on my Xeon server with 2x GTX 1080ti graphics cards.

The full settings used for training these models are captured in the log files and solver prototxt files. The initial training approach is outlined in the following table, and from this it can be seen that an attempt was made to train the models for ~1000 epochs each. This extended amount of training allows for exploratory options when picking a suitable model for deployment.

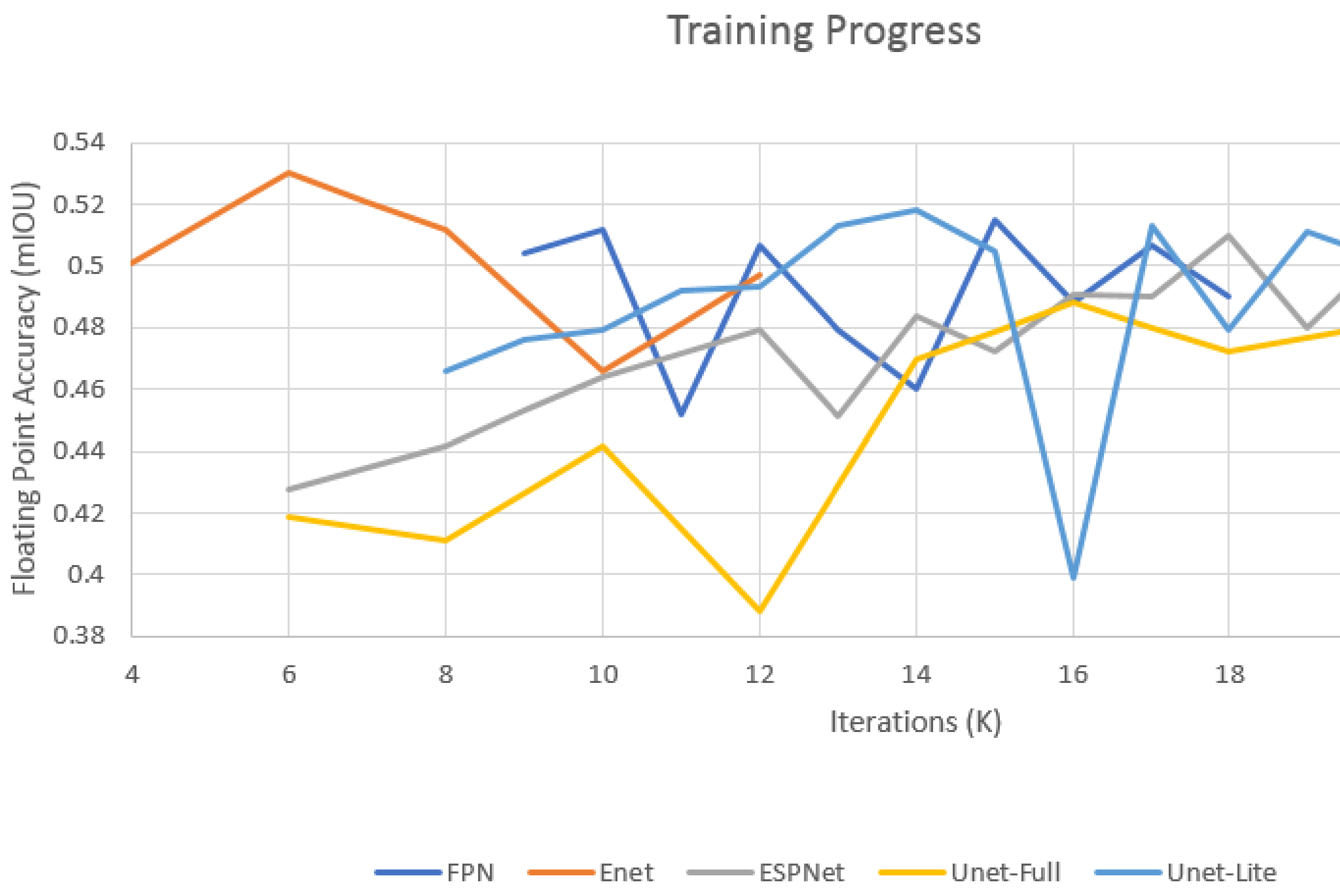


Model	Number of Iterations (encoder only)	Batch Size (encoder only)	Number of Iterations (full model)	Batch Size (full model)	Tot Estima Epoc
Enet	20000	50	12000	175	103
ESPNet	20000	50	12000	175	103
FPN	N/A	N/A	18000	170	102
Unet-Full	N/A	N/A	22000	100	73
Unet-Lite	N/A	N/A	18000	172	103

Each of the pre-trained models achieves different levels of accuracy in terms of mIOU and some of this variation is due to the training parameters used. An initial effort was made to keep the total training epochs around 1000 for each model and the effective batch size around 170-175, with the exception of Unet-full as it was an exceptionally large model, so a reduced batch size (and therefore number of epochs) was used to speed up the training process.

Note again that the intention of this tutorial is not to benchmark different models against each other, or even to show a model that works exceptionally well, but rather to show how different segmentation models can be trained, quantized, then deployed in Xilinx SoCs while maintaining the floating point model accuracy.

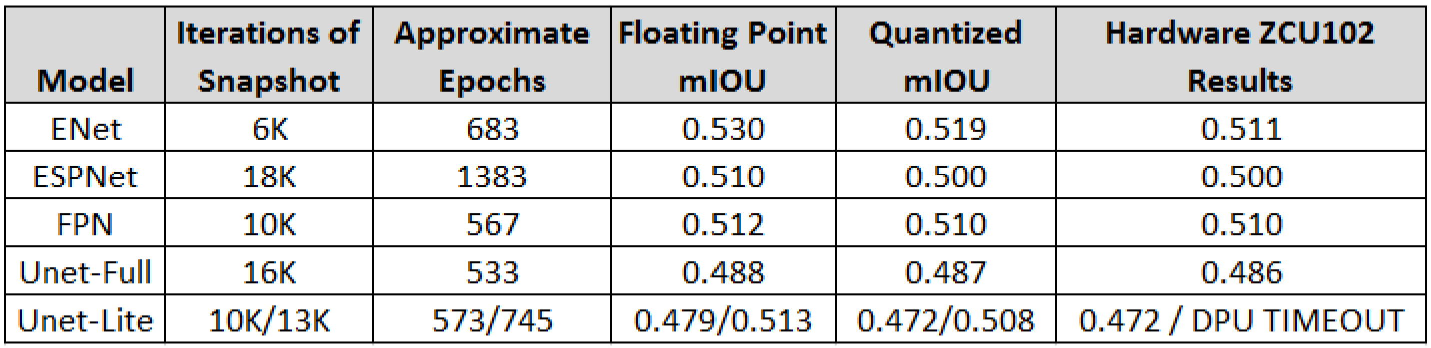
As the training progressed, regular mIOU measurements were taken using decent_q (don't worry if you don't understand this yet, it's covered in section 5 - part 3) to score the models against the Cityscapes validation dataset (500 images). When viewing the plot, recall again that ENet and ESPNet had separate encoder training, so the reduced number of iterations shown in this plot do not visualize that fact.



It can be seen from the plot that the model with the highest number of iterations does not necessarily correspond to the highest mIOU. You can also see from the fluctuations in mIOU that perhaps it might be possible to achieve better results by adjusting the learning rate and lr_policy, or by training some of the models for more iterations. In general, the models with the highest mIOUs were included as the pre-trained model for each respective network in this tutorial.

- For ENet -> 6K iteration model
- For ESPNet -> 18K iteration model
- For FPN -> 10K iteration model
- For Unet-Lite -> 10K/13K iteration models
- For Unet-Full -> 16K iteration model

Note that ESPNet continued to increase in mIOU at 12K iterations, so an additional 8K iterations of training were performed to find a higher mIOU model. Additional exploratory training was done for some of the other models as well, but the final models included as pre-trained are captured in the table below which shows the training snapshot used as well as the mIOU as was measured for the floating point model, the quantized model on the host machine, and the model deployed on the ZCU102 hardware. Again, don't worry if it isn't yet clear how these results were achieved. The latter sections in this tutorial explain how to measure the mIOU for each of these scenarios.



Model	Iterations of Snapshot	Approximate Epochs	Floating Point mIOU	Quantized mIOU	Hardware ZCU102 Results
ENet	6K	683	0.530	0.519	0.511
ESPNet	18K	1383	0.510	0.500	0.500
FPN	10K	567	0.512	0.510	0.510
Unet-Full	16K	533	0.488	0.487	0.486
Unet-Lite	10K/13K	573/745	0.479/0.513	0.472/0.508	0.472 / DPU TIMEOUT

The results as seen in the table are a bit confusing for Unet-lite, but essentially, an issue was encountered when using the 1024x512 images size for evaluating the 13K iteration model against the cityscapes validation dataset on the ZCU102 which caused an issue with the DPU timing out. The 10K model had lower accuracy in terms of mIOU, but did not exhibit this issue, so the 10K model was used for the evalution with a size of 1024x512, and the 13K iteration model is used with an input size of 512x256 when playing back the video as it produces better accuracy and works ok with the smaller input size. This issue is currently being investigated and the tutorial will be updated when a solution when one is found.

4.0 Evaluating the Floating Point Models on the Host PC

4.1 Evaluating the Models with Still Images and Displaying the Results

After training the models, it is useful to evaluate the model on the host PC before attempting to deploy it in the DPU. In order to evaluate the models, scripts have been provided under the Segment/workspace/scripts/test_scripts directory. Credit for these helpful scripts is given to Timo Saemann which he provided in his ENet tutorial. The scripts have been modified for the various tasks, but the base scripts from Timo Saemann have been used as a starting point for the evaluation code.

You can test both the encoder only models as well as the full models for all of the networks. To do so, execute the following steps.

1. Change directory to the `Segment/workspace/scripts/test_scripts` directory
2. The following scripts can be used to evaluate the various models on a single image:

- test_enet_encoder.sh
- test_enet.sh
- test_espnet_encoder.sh
- test_epsnet.sh
- test_fpn.sh
- test_unet-full.sh
- test_unet-lite.sh

The evaluation scripts contain paths to the caffemodel which will be evaluated. If you want to evaluate the pre-trained models that have been provided, you can simply execute them as is, otherwise, you will need to modify the path to the caffemodel and prototxt file. An example is shown below for enet which uses relative paths to point up to the model directory where the prototxt and trained snapshot exist.

```
export PYTHONPATH=../../../caffe-master/python
python test_enet.py \
--model ../../model/enet/deploy.prototxt \
--weights ../../model/enet/final_models/pretrained.caffemodel \
--colours ../cityscapes19.png \
--input ../munich_000000_000019_leftImg8bit.png \
--out_dir ./ --gpu 0
```

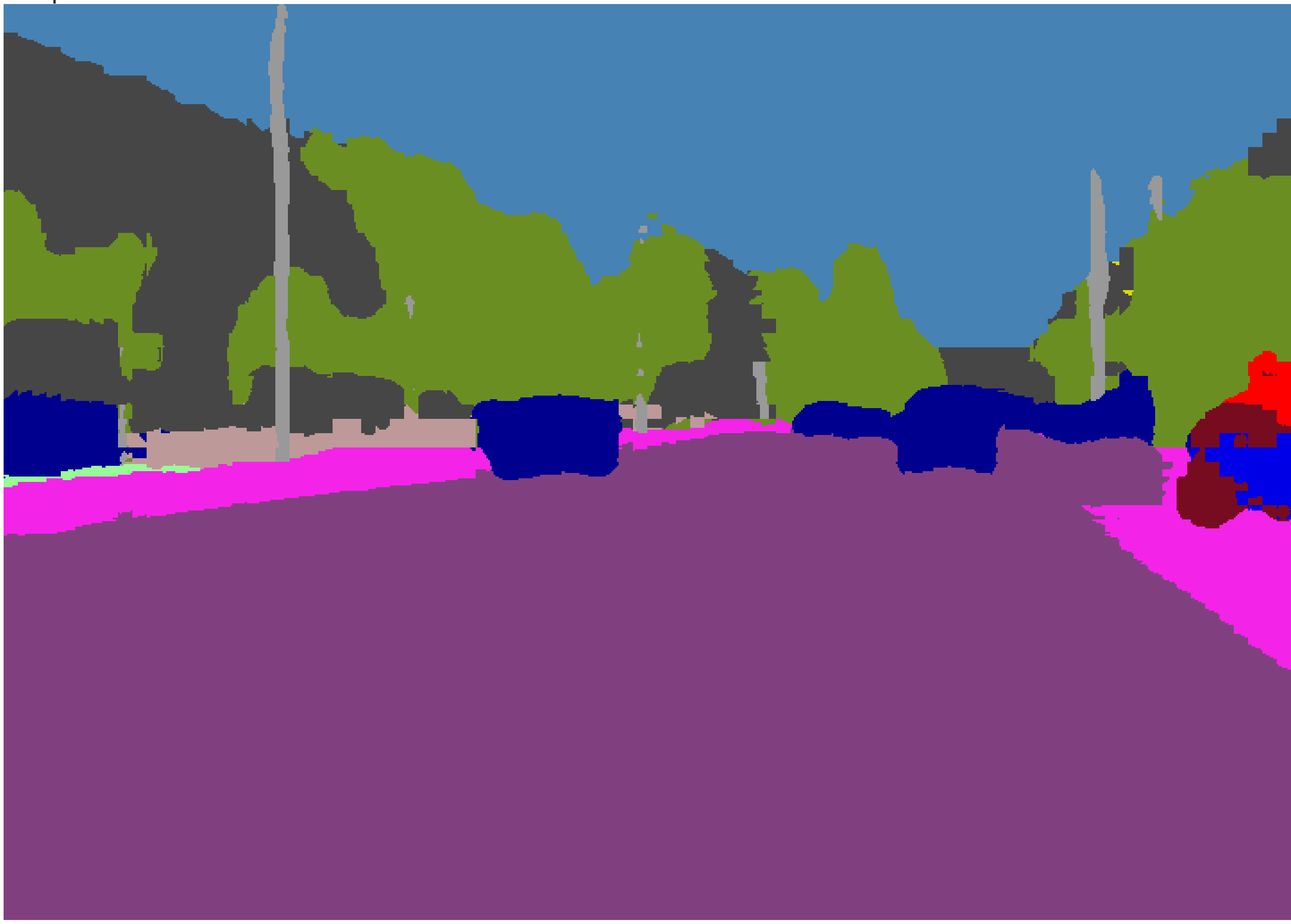
As can be seen from the screenshot above, these scripts will call the underlying python script which will run one forward inference on the trained model using the "munich_000000_000019_leftImg8bit.png" image as the input. The output will then color code the classes and display the output image. You could modify the python program to work on a recorded video or a webcam if desired, though the scope of this tutorial only provides an evaluation of a single images.

The input image and example outputs for the pre-trained networks can be seen below. Note the differences between the two networks as well the smoothing effect that including the full decoder stage has on the output.

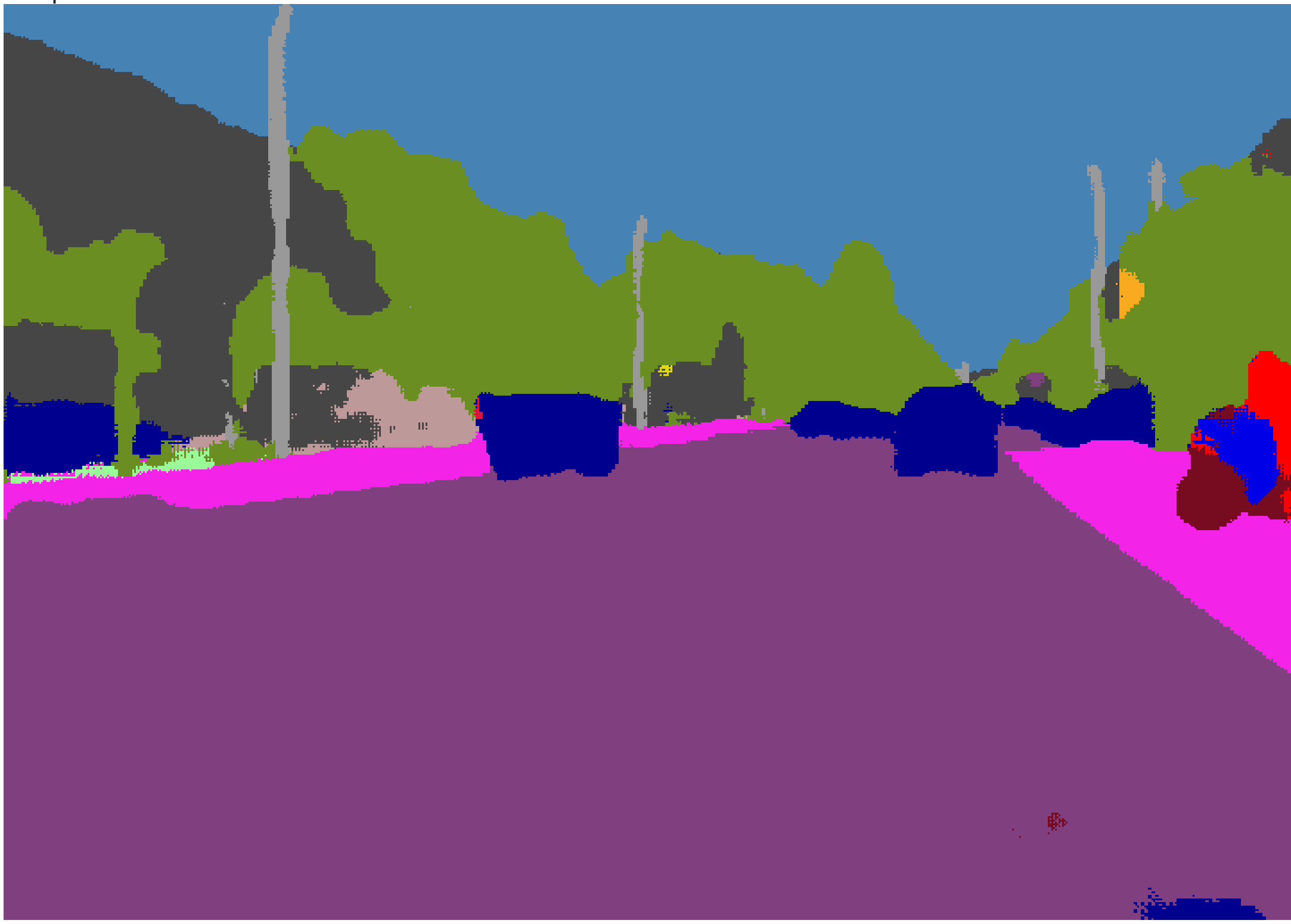
INPUT IMAGE FROM CITYSCAPES DATA:

HOTEL
Autohaus
Oberhaching

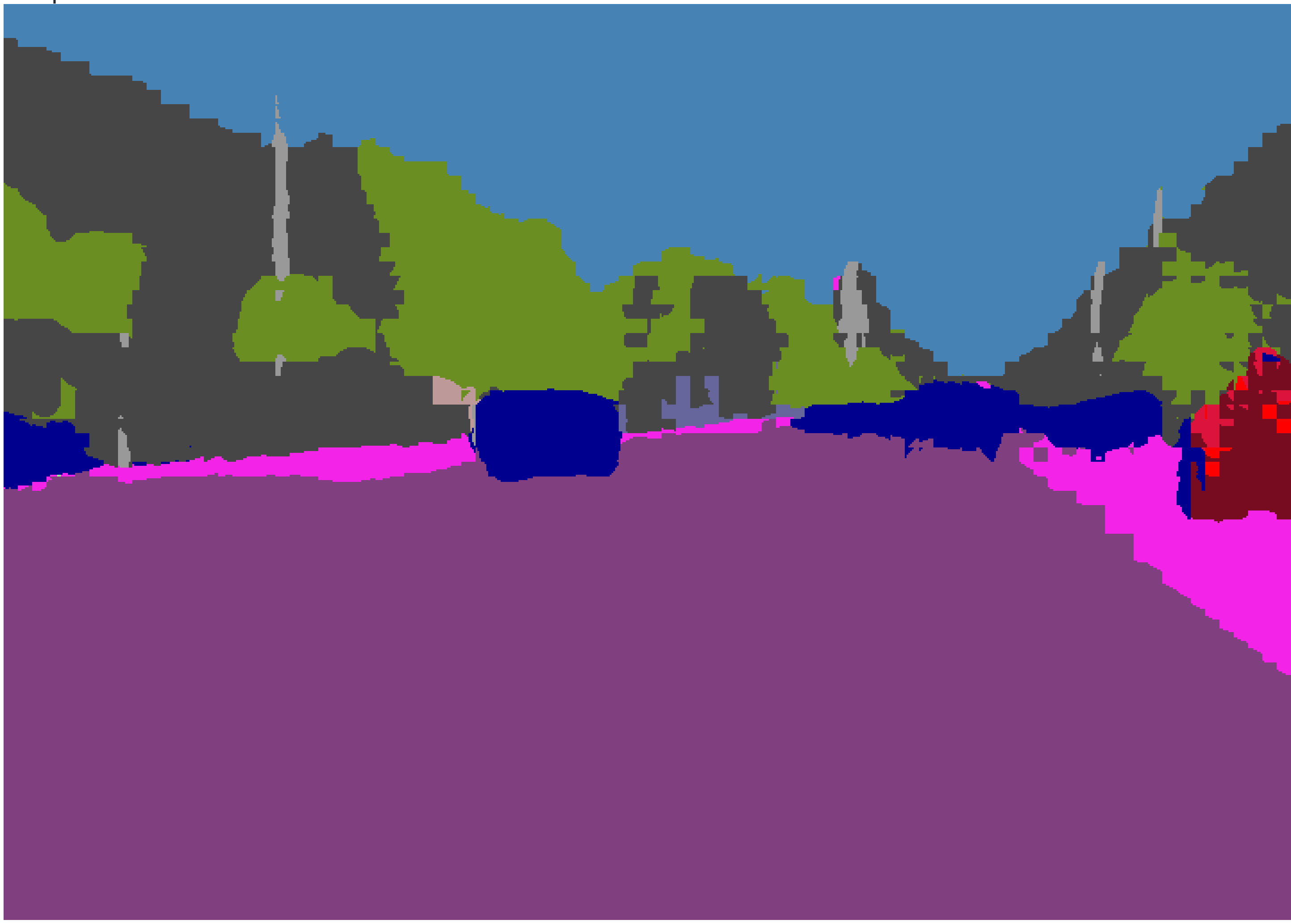
ENet Encoder Only
Output:



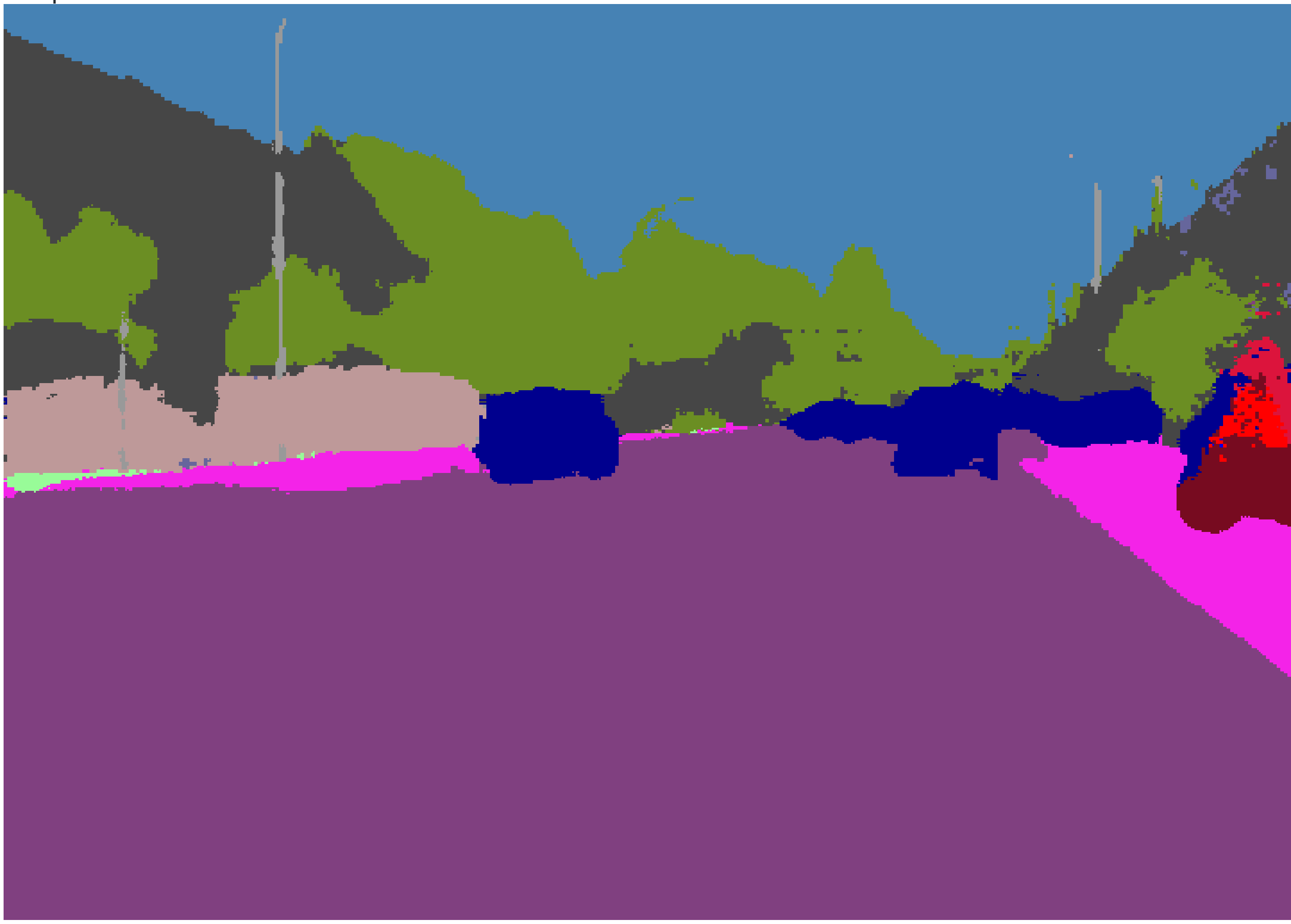
ENet Full Model
Output:



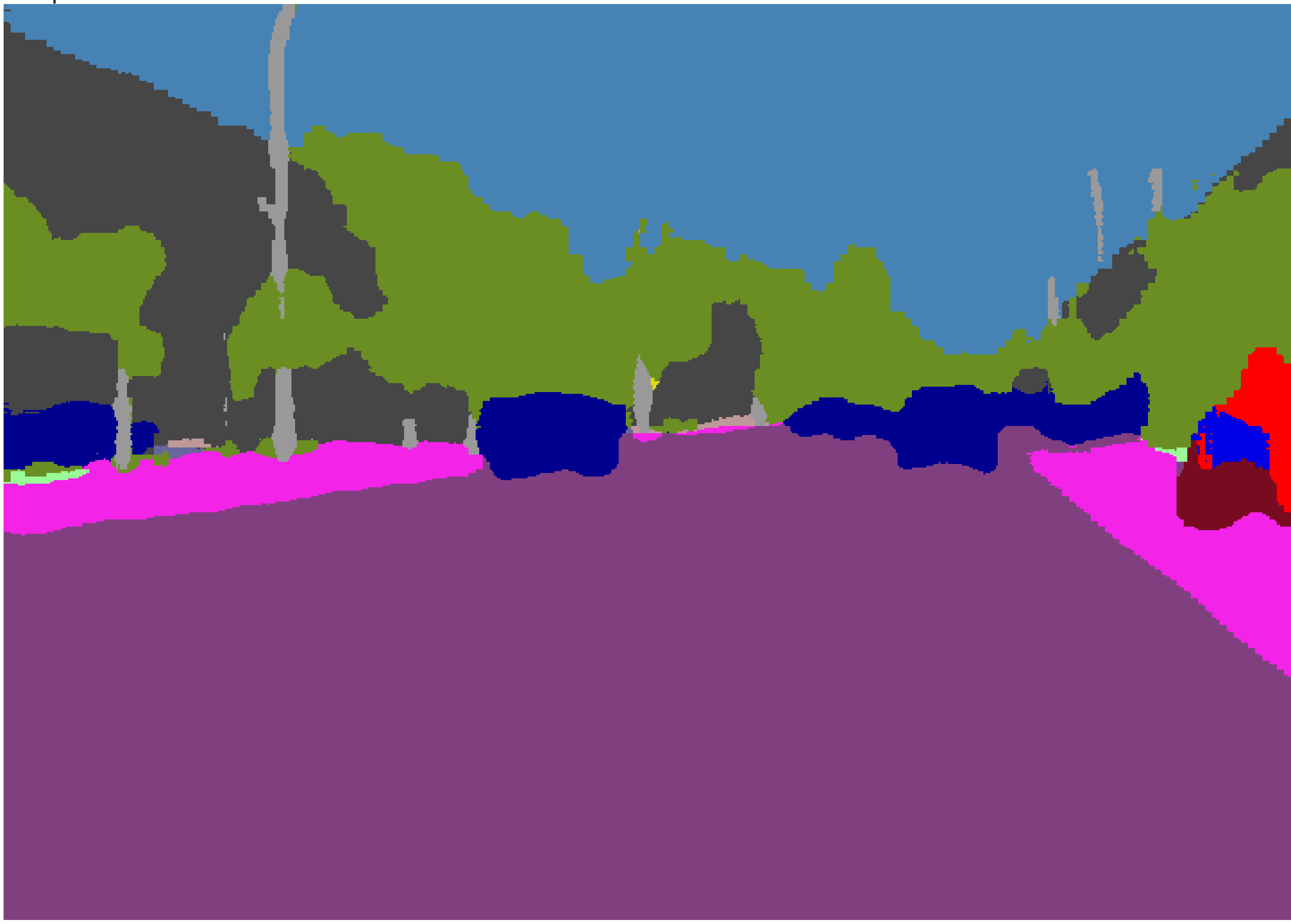
ESPNet Encoder Only
Output:



ESPNet Full Model
Output:

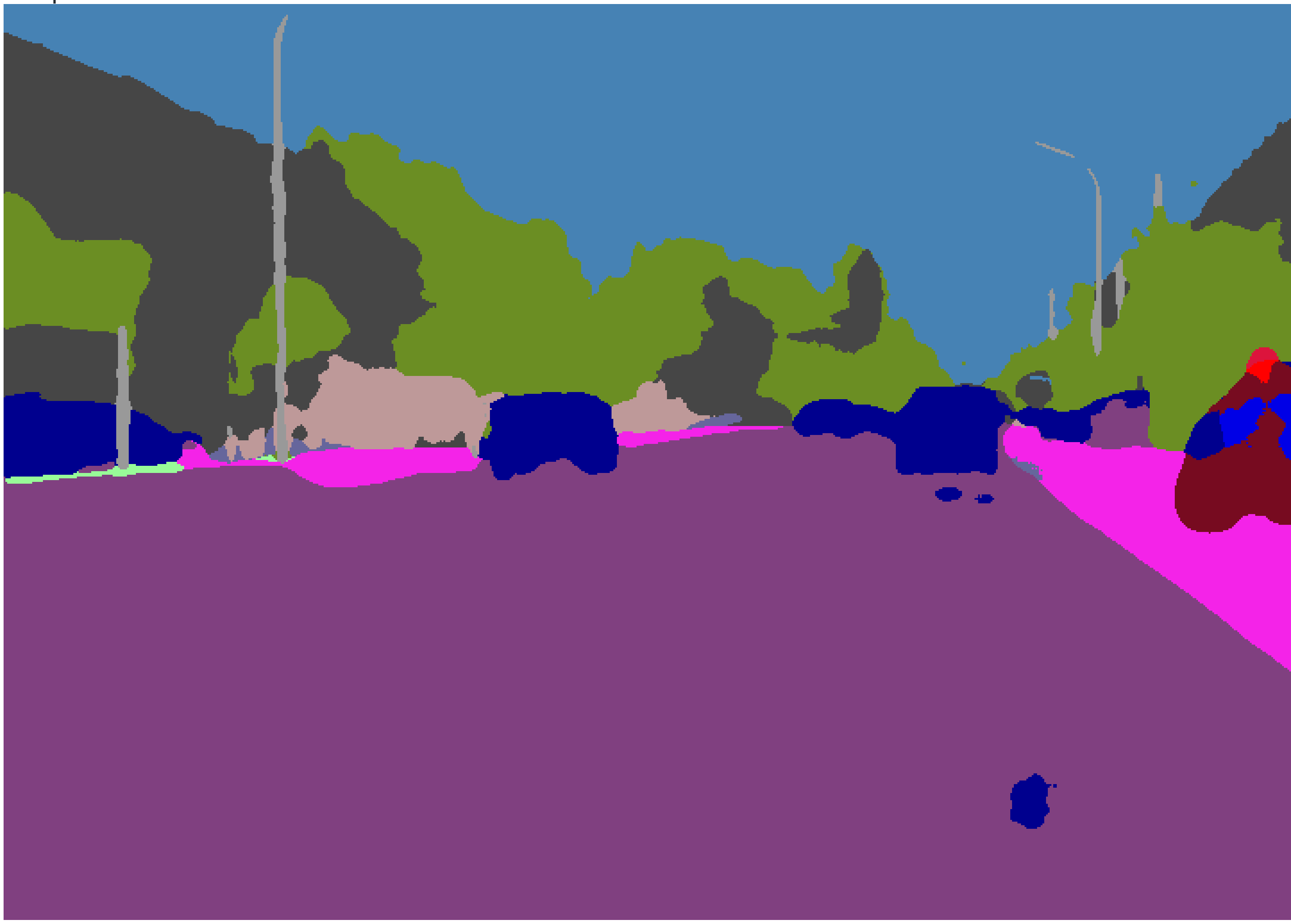


FPN Model
Output:

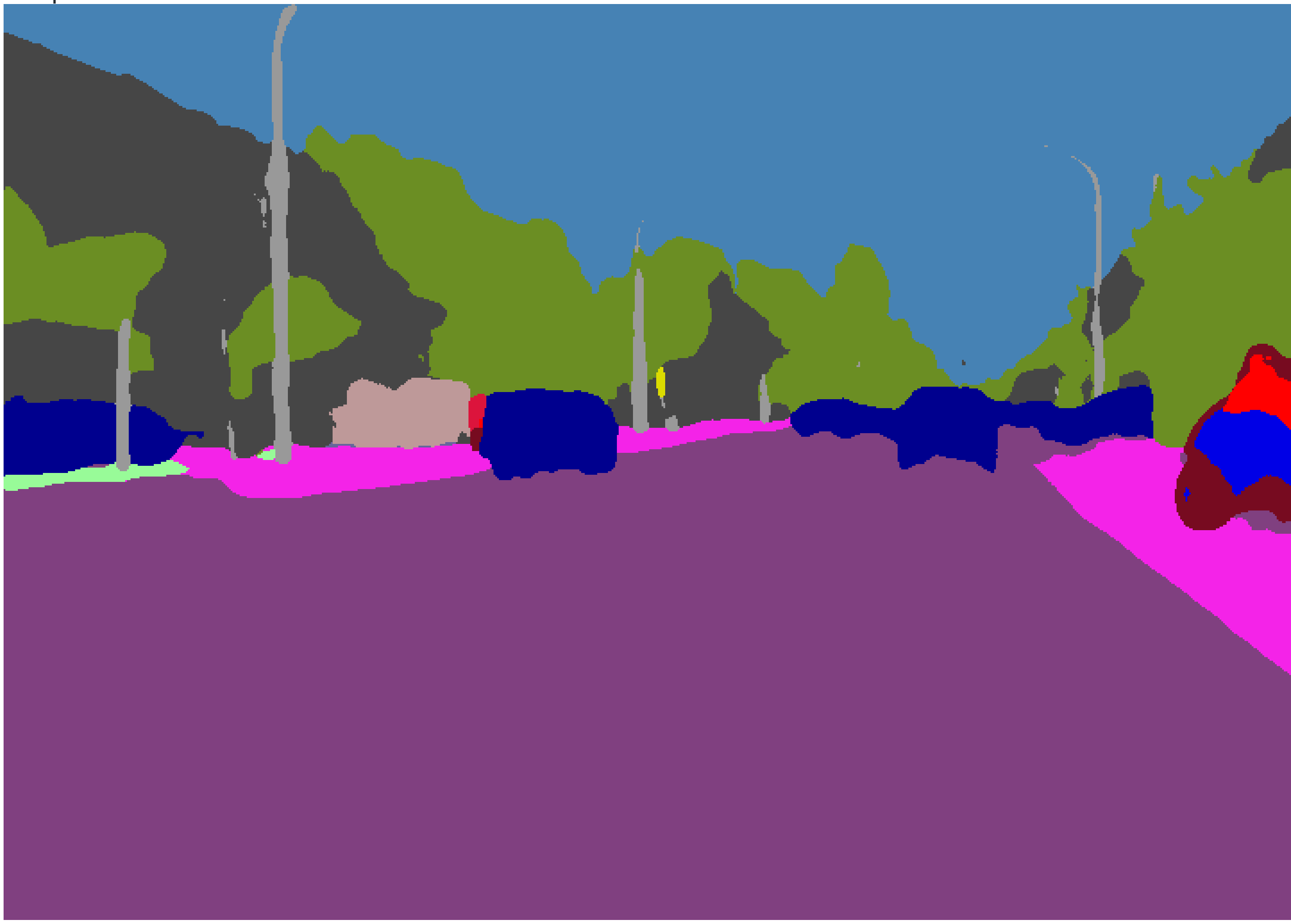


Unet-Full Model

Output:



Unet-Lite Model
Output:



If the output of your model looks reasonable, you can now proceed to step 5, however, there are other tests that may be performed at this point which are outlined in the remainder of this section.

It is also possible to test the inference time of the models on the host machine by running the following command from $CAFFE_ROOT:

For ENet Encoder Only:

```
./build/tools/caffe time \
-model ../workspace/model/enet/deploy_encoder.prototxt -gpu 0 -iterations 100
```

For ENet Full Model:

```
./build/tools/caffe time \
-model ../workspace/model/enet/deploy.prototxt -gpu 0 -iterations 100
```

For ESPNet Encoder Only:

```
./build/tools/caffe time \
-model ../workspace/model/espnet/deploy_encoder.prototxt -gpu 0 -iterations 100
```

For ESPNet Full Model:

```
./build/tools/caffe time \
-model ../workspace/model/espnet/deploy.prototxt -gpu 0 -iterations 100
```

My results for these various tests with a Xeon CPU and GTX 1080ti graphics card and a 512x256 input size for the models are as follows (note that the GPU power was estimated using the nvidia-smi command while running forward inference):

For ENet Encoder Only: Average Forward Pass 17 ms, 94W

For ENet Full Model: Average Forward Pass 27 ms, 93W

For ESPNet Encoder Only: Average Forward Pass 9.1 ms, 108W

For ESPNet Full Model: Average Forward Pass 10.3 ms, 108W

For FPN Model: Average Forward Pass 15.5 ms, 114W

For UNet-Full Model: Average Forward Pass 33 ms, 220W

For UNet-Lite Model: Average Forward Pass 14 ms, 176W

4.2 Measuring the Floating Point Model mIOU on the Cityscapes Validation dataset

In order to test the floating point and quantized model mIOU, the primary method is to use the decent_q test command. A local copy of decent_q is provided rather than the publicly distributed decent_q, as this local copy provides the capability to test both the floating point and quantized models.

In each of the model subdirectories under the Segment/DNNDK filepath, a script has been provided that will perform the following actions:

- Test the floating point model against the cityscapes validation dataset (run inference and calculate mIOU)
- Quantize the model using the calibration images from the Segment/DNNDK/data/cityscapes/calibration_imagesdirectory (Need to first

check step 5.0 on Quantization as directions are provided there for populating this directory).

To use this primary method, please proceed to the section 5.0 on quantization as some of the steps covered in that section will be needed to complete this step.

In addition to using the decent_q capabilities to test the floating point model, a couple secondary methods are included in this section. These methods employ the use of Caffe along with some python scripts to run forward inference on the floating point model and check the mIOU. There are two ways additional methods captured below to do this that I'm providing in this tutorial.

Both option 1 and option 2 use different python scripts to run forward inference and compare the mIOUs. The key differences between option 1 and option 2 python scripts are that option 2 uses 2048x1024 as the default input size (this is the native size of the cityscapes dataset samples) for the forward inference process whereas option 1 and decent_q uses 1024x512. If you modify the files to use 1024x512 for the input size for option 2, your results should be similar to option 1, but some differences may exist between all three approaches due to other variables such as methods used to resize the images.

Floating Point Model Test Using Python on Host Machine - Option 1

- Change directory to Segment/workspace/scripts/eval_option_1
- Make sure your CITYSCAPES_DATASET variable is exported properly to the location of dataset. If you have not done this, a default location will be used which will cause the script to fail.
- Next you need to prepare the validation images that will be used as a ground truth for comparison against the model output. This can be done by running the cls34_to_cls19.py script by entering `python -cls34_to_cls19.py`. This step only needs to be performed once for the ground truth images. Note that these will be stored in a folder called `test_gtFine_cls19` where your CITYSCAPES_DATASET is located.
- Next you need to run forward inference using the validation images. This can be done using the `forward_inference_model_name.sh` scripts provided (e.g. `./forward_inference_enet.sh`). These scripts will create a soft link to the pretrained caffe models in the working directory where the snapshots of the training process are stored. Then the script will run segmentation_miou_test.py to

perform forward inference using that model and store the results under the "results" directory.

- Finally, with the results captured and ground truth images prepared, run `./eval_segmentation.sh` script to compare the ground truth images and report the mIOU. The mIOU for all classes will be the first number reported and the other numbers reported will be the per class mIOU numbers.

Floating Point Model Test Using Python on Host Machine - Option 2

The cityscapesScripts that were downloaded in step 2.0 can also be used to perform mIOU measurements on the Cityscapes validation dataset. The process for this involves the following steps:

- Replace the `cityscapesScripts/cityscapesscripts/helpers/labels.py` with the modified labels.py which exits at `Segment/workspace/scripts/eval_option_2/cityscapes_modified_scripts/labels.py`. The key changes to this file involve modifying the classes to have only encodings 0-18 and 255 for ignored labels.

- Modify line 133 of cityscapesscripts/evaluation/evalPixelLevelSemanticLabeling.py

from:

```
args.groundTruthSearch  = os.path.join( args.cityscapesPath , "gtFine" , "val" , "*",
"*_gtFine_labelIds.png" )`
```

to

```
args.groundTruthSearch  = os.path.join( args.cityscapesPath , "gtFine" , "val" , "*",
"*_gtFine_labelTrainIds.png" )
```

- Reinstall the cityscapesScripts by changing directory into the cityscapesScripts folder then using pip:

- `sudo pip install .`

- make sure you have exported your CITYSCAPES_DATASET environment variable to point to the location of the cityscapes dataset. If you have not done this, a default location will be used which will cause the script to fail.

- Next change directory to where the provided scripts exist for the desired model (e.g. Segment/workspace/scripts/eval_option_2/enet)

- Open the script validate_*.sh (e.g. validate_enet.sh).

- In this script you can see that the python script validate_enet.py is called 3x, once for each subfolder of the validation images provided by cityscapes. Credit for these helpful scripts is given to Timo Saemann which he provided in his ENet tutorial. The scripts have been modified for the various tasks, but the base scripts from Timo Saemann have been used as a starting point for the forward inference code.
- If there are any paths that need to change in this script, you can make modifications to point to the correct caffemodel and prototxt file. Local prototxt files have been stored in each of these directories such that you can make changes to the input size as this will affect accuracy.
- The default input size should be 2048x1024 as this matches the cityscapes dataset annotations. If you want to try testing with other input sizes, you can use the validate_enet_resize.sh which will use the deploy_enet_resize.prototxt.
- Note also that the Unet-full model does not have a resize script because its input size is 1024x512. This is because the model is too large to run inference on with the provided version of Caffe, so a reduced input size is used to test the model.
- After the forward inference is run, results are stored under the "results" directory, after which the csEvalPixelLevelSemanticLabeling python method is called to compare these results to the cityscapes validation data ground truths.

5.0 Quantizing and Compiling the Segmentation networks for DPU implementation

5.0 PART 1: Installing the DNNDK tools

In section 1, we downloaded the DNNDK tools and copied them over to the ZCU102 board. A portion of these tools also needs to be installed on the host x86 machine for quantizing and compiling the model.

The tools needed are contained under the host_x86 tools directory.

1. Please copy the host_x86 folder and its subdirectories to the host machine.

2. Next cd into the host_x86 directory and install the host tools.

```
sudo ./install.sh ZCU102
```

NOTE: The target for this tutorial is the ZCU102, but it should be possible to target other boards as well by changing the target shown above when installing the tools and also modifying the dnnc command to target the correct DPU. As a quick reference, the ZCU102 and ZCU104 use a 4096FA DPU.

Please refer to the DNNDK User Guide for more details on the DNNDK tools.

If you would like to quantize and deploy the model, continue onto 5.0 part 2, otherwise if you would like to first test the quantized and floating point models and compare the mIOU between the two, then jump down to 5.0 part 3.

5.0 PART 2: Configuring the Files for Quantization, Compilation, and mIOU Testing:

1. I have included an example workspace in Segment/DNNDK to show how the DNNDK tools may be invoked as well as the necessary modifications to the prototxt files for both quantization/compilation and testing the float and quantized model mIOUs. Change directory to the DNNDK directory before proceeding to the next step.

2. Within the DNNDK directory, there is a subdirectory for each model. Inside each model directory several files:

- "float.prototxt" is used for quantizing/compiling the models for deployment on the target hardware
- "float_test.prototxt" is used for testing the float and quantized models to report the mIOU against the cityscapes validation dataset
- "float.caffemodel" is the pre-trained caffemodel.
- "quantize_and_compile.sh" is a script that is used to perform both quantization and compilation (decent_q and dnnc) for deployment on the target hardware
- "test_float_and_quantized.sh" is a script that will test both the floating point and quantized models and report out the mIOU for each

- There is also a subdirectory for decent as a local copy of decent_q is provided rather than the publicly distributed decent_q, as this local copy provides the capability to test both the floating point and quantized models.

3. Open the "float.prototxt" that is included as an example in the DNNDK subfolders (i.e. ENet, ESPNet, etc.).

The "float.prototxt" files should be mostly identical to your "train_val.prototxt" except for the following:

- The input layer has changed from "ImageSegData" type to "ImageData"
- Paths have been specified to the calibration data in a relative fashion so that they point to the correct locations if the directory structure is left intact.
- Note by default that the prototxt files are set to generate a 512x256 input size model which is intended for use with the xxx_video applications (e.g. fpn_video). If you wish to run the evaluation in hardware on cityscapes validation images rather than on the recorded video (e.g. fpn_eval), the applications use 1024x512, so you will need to modify these input layers accordingly (the float_test.prototxt files have the input set for 1024x512 if you wish to use this as an example).

```
line 11:  source: "../data/cityscapes/calibration.txt"
line 12:  root_folder: "../data/cityscapes/calibration_images/"
```

- The "SoftmaxWithLoss" layer has been changed to "SoftMax" and the "Accuracy" layer has been removed. These layers were previously used to compute loss and accuracy for the training phase, so they have now been updated for deployment.

Important note for ENet float.prototxt: the "UpsamplingBilinear2d_x" layers have been changed to "DeephiResize" type because decent doesn't support bilinear upsampling with the deconvolution layer

You can use these prototxt files directly if the differences mentioned above are the only deltas between your train_val.prototxt file and float.prototxt. Otherwise, if you are deploying the encoder model only or a modified version, you will need to update your train_val.prototxt to accommodate for the differences mentioned above, rename that file to "float.prototxt", and place it in the correct directory.

4. The calibration data needs to be populated into the Segment/DNNDK/data/cityscapes/calibration_images directory. This data consists of a list of images which are specified in the calibration.txt file, and 1000

test images from Cityscapes. These will be used by the `decent quantize` process as stimulus for calibration of the model dynamic range during the quantization process.

The data listed in the calibration.txt file calls out the following 1000 images:

- the first 100 images from CITYSCAPES_DATASET/leftImg8bit/test/berlin
- all images from $CITYSCAPES_DATASET/leftImg8bit/test/bielefeld
- all images from $CITYSCAPES_DATASET/leftImg8bit/test/bonn
- all images from $CITYSCAPES_DATASET/leftImg8bit/test/mainz
- the first 373 images from $CITYSCAPES_DATASET/leftImg8bit/test/munich

You will need to copy these images or potentially create soft links from the dataset directories listed about to the Segment/DNNDK/data/cityscapes/calibration_images directory. You can use other calibration images if desired, however, the provided calibration.txt file uses the images listed above.

5. Next copy your latest trained model from Caffe into the **Segment/DNNDK/model_subdirectory_name** directory (or reuse the already populated float.caffemodel) and rename it "float.caffemodel". This model should be located wherever the snapshot was saved from the the training step.

6. Next run the quantization tools using the following command:

```
./quantize_and_compile.sh
```

If you open the script, you will see the following contents which indicate several things - first of all, you should make sure the GPUID environment variable is set correctly for your machine. If you have only one GPU, this should be '0', otherwise, please change this to the index for the desired GPU to use for quantization.

Secondarily, there is a Segment/DNNDK/decent/setup_decent_q.sh script being called which checks your nVidia environment and uses the correct local decent_q executable for quantization. The reason for this is that at the time this tutorial was authored, the public version of decent did not yet have the capability to perform testing on the floating point models, so this version of decent_q has been provided with this tutorial to enable mIOU testing for both the floating point and quantized models.

Next, you can see that `decent_q_segment quantize` is called with various arguments including calibration iterations, GPUID, paths to the input and output models, and a tee to dump the output to a text file in the decent_output directory.

For reference, I have included an enet decent log file and espent decent log file that shows the output of my console after running the decent command. You should see something similar after running the command on your machine.

Finally, the `dnnc` command is called which compiles the floating point model and produces a file called "dpu_segmentation_0.elf" under the dnnc_output directory.
For reference, I have included an enet dnnc log file and espent dnnc log file that shows the output of my console after the dnnc command is run. You should see something similar after running the command on your machine.

```
#!/usr/bin/env bash
export GPUID=0
net=segmentation
source ../decent/setup_decent_q.sh

#working directory
work_dir=$(pwd)
#path of float model
model_dir=decent_output
#output directory
output_dir=dnnc_output

echo "quantizing network: $(pwd)/float.prototxt"
./../decent/decent_q_segment quantize            \
          -model $(pwd)/float.prototxt     \
          -weights $(pwd)/float.caffemodel \
          -gpu $GPUID \
          -calib_iter 1000 \
          -output_dir ${model_dir} 2>&1 | tee ${model_dir}/decent_log.txt

echo "Compiling network: ${net}"

dnnc    --prototxt=${model_dir}/deploy.prototxt \
        --caffemodel=${model_dir}/deploy.caffemodel \
        --output_dir=${output_dir} \
        --net_name=${net} --dpu=4096FA \
        --cpu_arch=arm64 2>&1 | tee ${output_dir}/dnnc_log.txt
```

At this point, an elf file should have been created in the **dnnc_output** directory which can be used in the final step which is to run the models on the ZCU102. If desired, you can also proceed to the Part 3 of 5.0 which is testing the floating point and quantized models.

5.0 PART 3: Testing the Floating Point and Quantized Models

As mentioned in the previous section, files have been provided under the `Segment/DNNDK/model_subdirectory_name` filepath which can enable you to rapidly test the mIOU of both the floating point model as well as the quantized model on the cityscapes validation dataset. In order to perform this testing, perform the following steps:

1. Open the `Segment/DNNDK/data/val_img_seg_nomap.txt` file with a text editor.

2. Notice that this file contains paths to the cityscapes validation dataset as they are stored on my local machine. The left column has a path to the input image, and the right column has a path to the labels. You need to modify the root directory portion of both paths to point to the location of the cityscapes dataset on your machine.

3. Open the float_test.prototxt file that corresponds to the model of interest. Notice that there are several differences between this file and the float.prototxt that was used for deployment. The reason for this is that the DeephiResize layer causes some problems in the current version of decent which will prevent dnnc from compiling the model (it causes the input layer to be renamed to "resize_down" which causes dnnc to fail- for this reason two separate files are used, one for testing and one for deployment).

The new additions to this model are to support the auto_test and test decent_q commands:

- The input size of the model has been changed from 512x256 to 1024x512. This is because the larger input size produces better mIOU results. It would be possible to use other sizes such as the native input size for the citysacpes dataset which is 2048x1024, but testing the models would take longer and the Unet-full model will not work in this case because of some limitations on the Caffe distribution used within the decent_q tool. Additionally, the models were trained with an input crop size of 512, so it is not necessarily expected that using the larger size will produce better results.

- An additional input layer "ImageSegData" has been added which has a path to the val_img_seg_nomap.txt file. This is how the labels and input images are supplied for the testing procedure.

- A layer after this called "resize_down" has been added to scale the input image to the desired input size for the model (in this case 1024x512).

- A new layer at the end of the model has been added called "SegmentPixelIOU" which is a custom caffe layer packed up within the decent_q tool. If you noticed, the val_img_seg_nomap.txt file actually points to the *gtFIne_labelIds* rather than *gtFine_labelTrainIds*. This is because the SegmentPixelIOU layer has been coded to automatically relabel the classes from the cityscapes labels such that the classes match gtFine_labelTrainIds and values 255 are ignored.

4. Open the one of the `test_float_and_quantized.sh` scripts. The contents of this script are shown below. You will only need to edit the GPUID to specify the correct GPU index for your tests. Note that the log files will be captured under the test_results subdirectory for both the floating point and quantized results.

```
export GPUID=0
export WKDIR=`pwd`
cd ../decent
source setup_decent_q.sh
cd $WKDIR
./../decent/decent_q_segment test -model float_test.prototxt -weights
float.caffemodel -test_iter 500 -gpu $GPUID 2>&1 | tee
test_results/float_model_test.txt

#working directory
work_dir=$(pwd)
#path of float model
model_dir=${work_dir}
#output directory
output_dir=${work_dir}/decent_output

./../decent/decent_q_segment quantize             \
          -model ${model_dir}/float_test.prototxt \
          -weights ${model_dir}/float.caffemodel  \
          -gpu $GPUID \
          -calib_iter 1000 \
          -test_iter 500 \
          -auto_test \
          -output_dir ${output_dir} 2>&1 | tee test_results/quantized_model_test.txt
```

5. Execute the Script by running the following command. This will take some time which varies depending on the GPU hardware available as well as which model is being run. I have included example test results from a previous run under the associated model directories such as Segment/DNNDK/FPN/test_results. Note that the previous run results I have included does not necessarily represent the best performing snapshot - it is just an example of the output of running the test script.

```
./test_float_and_quantized.sh
```

At this point, the quantized and floating point models have been fully verified on the host and you are ready to proceed to deploying the models to the target hardware, however, if you skipped the section on pre-trained models you may be wondering how they scored. Jump back up 3.1.0 About the Pre-Trained Models to see the results.

6.0 Running the Models on the ZCU102

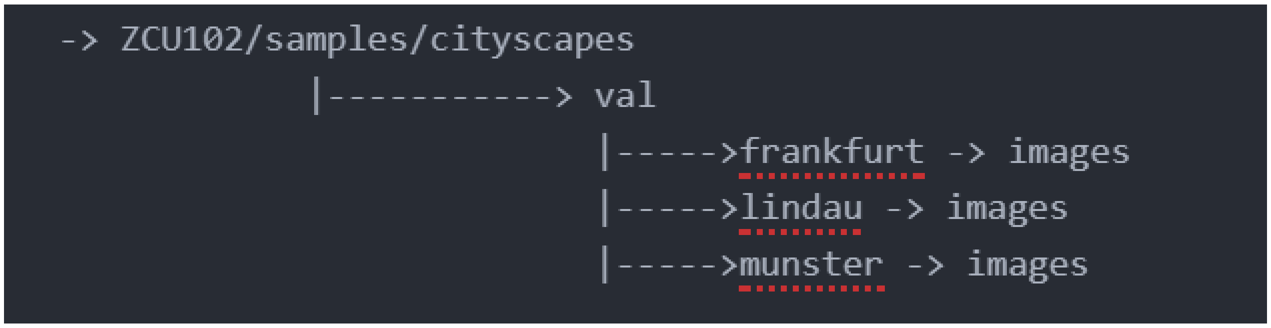
The final step in this tutorial is to run the models on the target hardware - in my case, a ZCU102, but other development boards may also be used (just make sure you changed the dnnc command when compiling the model to target the correct DPU associated with your board). In order to expedite the process for running on the target hardware, I have included software applications for each model that allow you to perform two different types of tests:

- Test the model using a video file and display the output. By default, this uses a 512x256 input size, and a video file titled "traffic.mp4" has been included under ZCU102/samples/video directory. The software application in the "model_name_video" (e.g. enet_video) subfolder can be used to test the model with this video file.

- Run forward inference on the model using the cityscapes validation dataset and capture the results for post processing back on the host machine. This uses a 1024x512 input size by default to match the mIOU tests that were done on the host previously, but other input sizes may also be used. The results of running this software application will be captured under the `software/samples/model_name/results` folder. The software application in the "model_name_eval" (e.g. enet_eval) subfolder can be used to perform this test.

In order to run the tests, it is assumed that you have a ZCU102 revision 1.0 or newer connected to a DisplayPort monitor, keyboard and mouse, and an Ethernet connection to a host machine as described in step 1.1. Then perform the following steps:

1. Copy the cityscapes validation images into the `ZCU102/samples/cityscapes` folder on your host machine. After this you should have a directory structure that looks like the following:



```
 -> ZCU102/samples/cityscapes
              |-----------> val
                                 |----->frankfurt -> images
                                 |----->lindau -> images
                                 |----->munster -> images
```

2. At this point you can either copy over your compiled model .elf file (output of dnnc) into the model subdirectory such

as software/samples/enet_eval/model/dpu_segmentation_0.elf, or use my pre-populated models that have already been included in that directory. Note that the makefile assumes the name of the .elf file is "dpu_segmentation_0.elf" and it also assumes that you used the name "net=segmentation" when compiling the model with dnnc. You can see this in the software/samples/enet_eval/src/main.cc files for each of the associated applications.

3. Boot the ZCU102 board
4. Launch the Linux terminal using the keyboard/mouse connected to the ZCU102
5. Configure the IP address for a compatible range with your host machine (I use 192.168.1.102 and then set my laptop to 192.168.1.101). The command needed to perform this step is:

```
ifconfig eth0 192.168.1.102
```

6. Launch a file transfer program - I like to use MobaXterm, though others such as pscp or WinSCP can also be used.
7. Transfer the entire samples directory over to the board
8. Change directory into one of the video based examples such as 'enet_video'
9. Run `make clean`, then `make -j4` to make the sample application. This will compile the ARM application and link in the DPU model (located under the models subdirectory) with the ARM64 executable to run on the A53 cores.
10. To execute the application, use the following command:

**for the _video applications*

```
./segmentation ../video/traffic.mp4
```

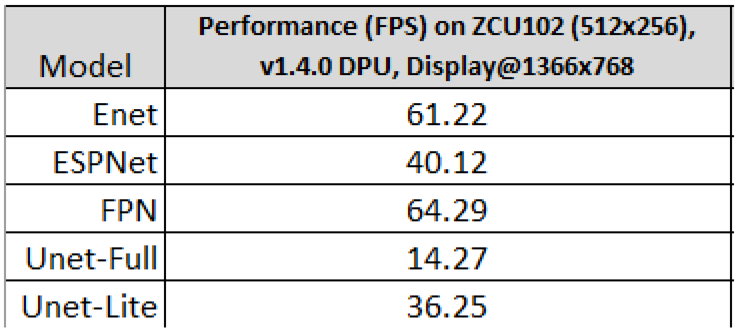
**for the _eval applications*

```
./evaluation
```

- When running the video application, you should see the video play back with overlays that color code the various classes from the scene onto the monitor. After the playback is completed, an FPS number will be reported in the terminal which summarizes the forward inference performance from the hardware run.

- When running the evaluation application, you should see a list of the validation images on the console being read into DDR memory, then the text ***Processing***, after which there will be delay while forward inference is run on the images. The final output will be 500 images with the class labels for pixel values stored into the results directory. These images can then be post processed back on the host machine to extract the mIOU score for the model.

I have observed the following performance using the pre-trained models on the ZCU102:

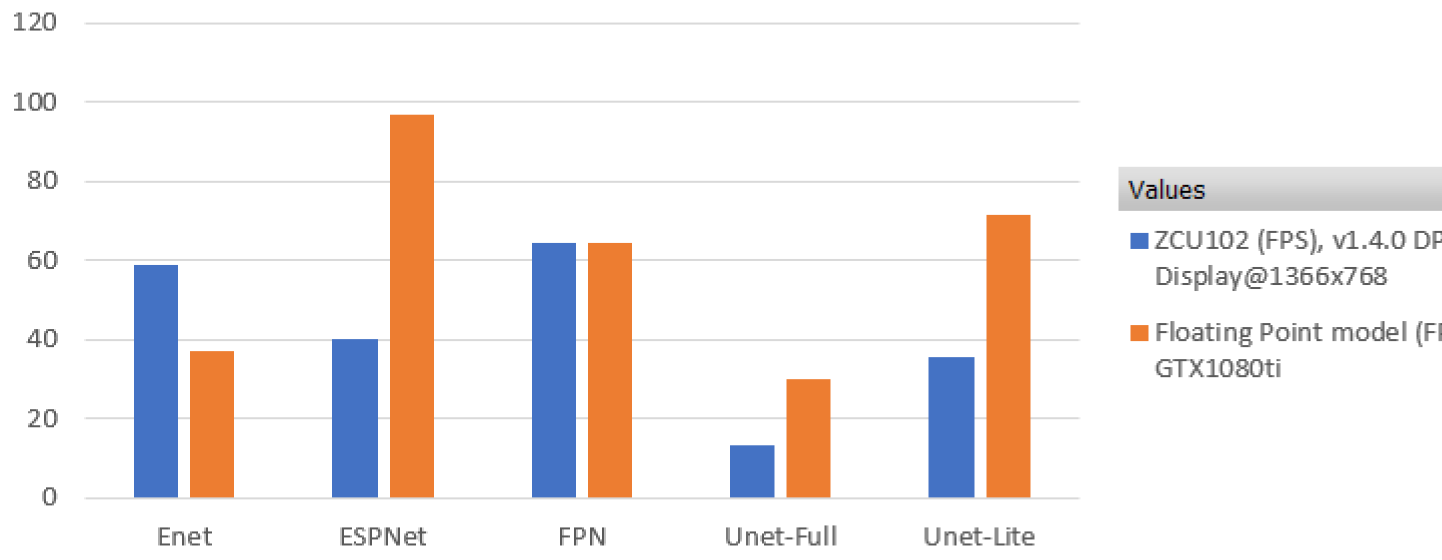


Model	**Performance (FPS) on ZCU102 (512x256), v1.4.0 DPU, Display@1366x768**
Enet	61.22
ESPNet	40.12
FPN	64.29
Unet-Full	14.27
Unet-Lite	36.25

Using data gathered throughout this tutorial, we can compare the performance of the ZCU102 vs. the GTX1080ti graphics card that was used to time the models from section 4.1. Albeit, this isn't a fair comparison for two reasons:

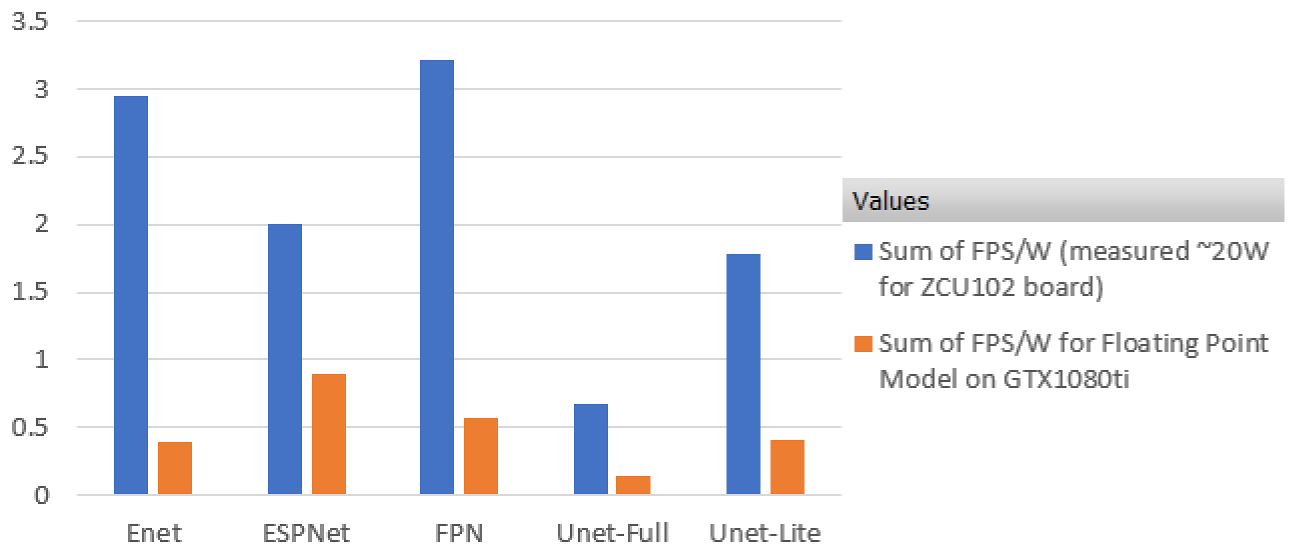
1. We are comparing an embedded ~20W device with a 225W GPU
2. The ZCU102 execution time includes reading/preparing the input and displaying the output whereas the GPU measurement only includes the forward inference time of the models

That said, this still provides some data points which are useful to garner further understanding. The following chart shows a comparison between the FPS as measured on the ZCU102 vs. the GTX1080ti.

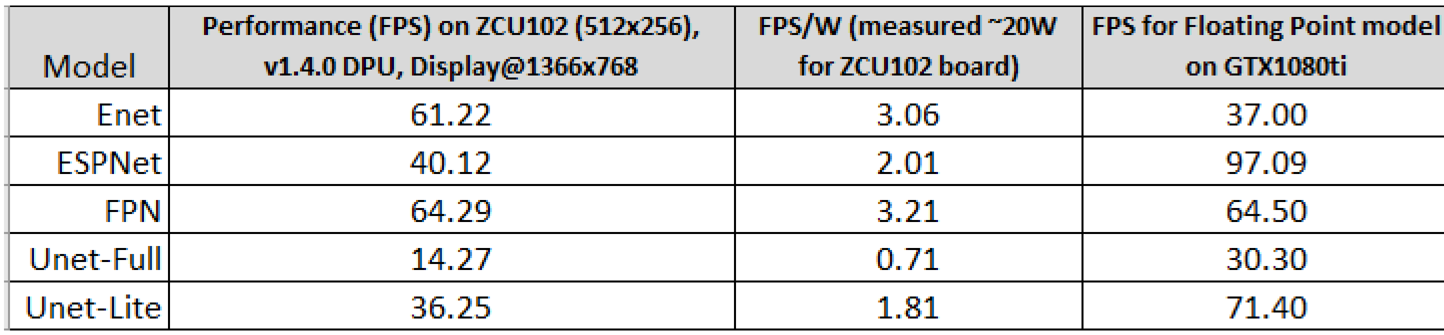


What is perhaps a bit more useful than comparing raw FPS, however, is to compare FPS/W (performance/Watt) as this is a more generic comparison of what performance is achievable for a certain power cost. Bear in mind, this is still not a fair comparison due to reason 2, but the value of a Xilinx SoC starts to shine a little more in this light. In reality the advantage is even more pronounced if only the DPU throughput is considered.

In order to perform this comparison, ~20W was measured on the ZCU102 board during forward inference, and the nvidia-smi tool was used to read the power during forward inference of each of the models as part of section 4.1. The comparison between the two can be seen in the following figure.



The same data can be seen in table format below:



Model	Performance (FPS) on ZCU102 (512x256), v1.4.0 DPU, Display@1366x768	FPS/W (measured ~20W for ZCU102 board)	FPS for Floating Point model on GTX1080ti
Enet	61.22	3.06	37.00
ESPNet	40.12	2.01	97.09
FPN	64.29	3.21	64.50
Unet-Full	14.27	0.71	30.30
Unet-Lite	36.25	1.81	71.40

At this point, you have verified the model functionality on the ZCU102 board and the only step left is to post process the images if you ran the evaluation software. The mIOU score from this can then be compared the mIOU that decent_q measured previously on the host machine. In order to do this, proceed to the final step, "7.0 Post processing the Hardware Inference Output".

7.0 Post processing the Hardware Inference Output

In order to post process the output of the hardware run, scripts have been provided under the Segment/workspace/scripts/evaluation/postprocess folder.

The following steps can be performed to complete this process:

- Copy the results folder from the target hardware back to the host machine into the Segment/workspace/scripts/evaluation/postprocess folder.
- Make sure your $CITYSCAPES_DATASET variable is exported properly to the location of dataset. If you have not done this, a default location will be used which will cause the script to fail unless your locations matches what is used as the default in the script.
- Next you need to prepare the validation images that will be used as a ground truth for comparison against the model output. This can be done by running the cls34_to_cls19.py script by entering `python -cls34_to_cls19.py`. This step only needs to be performed once for the ground truth images. If you already completed this as part of section 4.2, you can skip this step. Note that these will be stored in a folder called `test_gtFine_cls19` where your $CITYSCAPES_DATASET is located.
- Now run the eval_segmentation.sh script by entering `./eval_segmentation.sh`.

The output of this step should be a list of IOUs starting with the mIOU for all the classes (this is the number to compare to the decent_q quantized model mIOU). The other numbers are per clas IOU numbers for the validation dataset. I already completed this step for the pre-trained models and you can refer back to section "3.1.0 About the Pre-Trained Models" to see the results.

Summary

Looking back, we've covered a lot of ground, including walking through the process of preparing, training, testing, and deploying 5 different segmentation models. The goal of this tutorial was not to show a perfectly optimized solution, but rather to blaze a trail so you experts and explorers can streamline your own segmentation model development and rapidly deploy those models on a Xilinx SoC/MPSoC.

The beauty of this solution is that there is a full portfolio of Zynq-7000 and Zynq Ultrascale+ devices (qualified for commercial, industrial, automotive, and aerospace and defense end markets) and various DPU configurations that allow you to scale for low power applications that that require < 3W as well as dial up the performance for higher end products (where you may need hundreds to thousands of FPS), such as deployments on PCIe accelerator class products such as Alveo.

All of this is possible while using the same design approaches and tool flow, without re-inventing algorithms for low end vs. high end products. You can even make trade-offs for DPU size vs. traditional pre/post-processing hardware acceleration (e.g. optical flow, stereo block matching, scaling, de-interlacing, FFTs, or even custom Image Sensor Pipelines). The number of potential implementations are virtually endless, so you can truly build an optimal solution for your application that maximizes your differentiation.