Creating and training a U-Net model with PyTorch for 2D & 3D semantic segmentation: Dataset building [1/4]

A guide to semantic segmentation with PyTorch and the U-Net

Johannes Schmidt Dec 2, 2020 · 10 min read

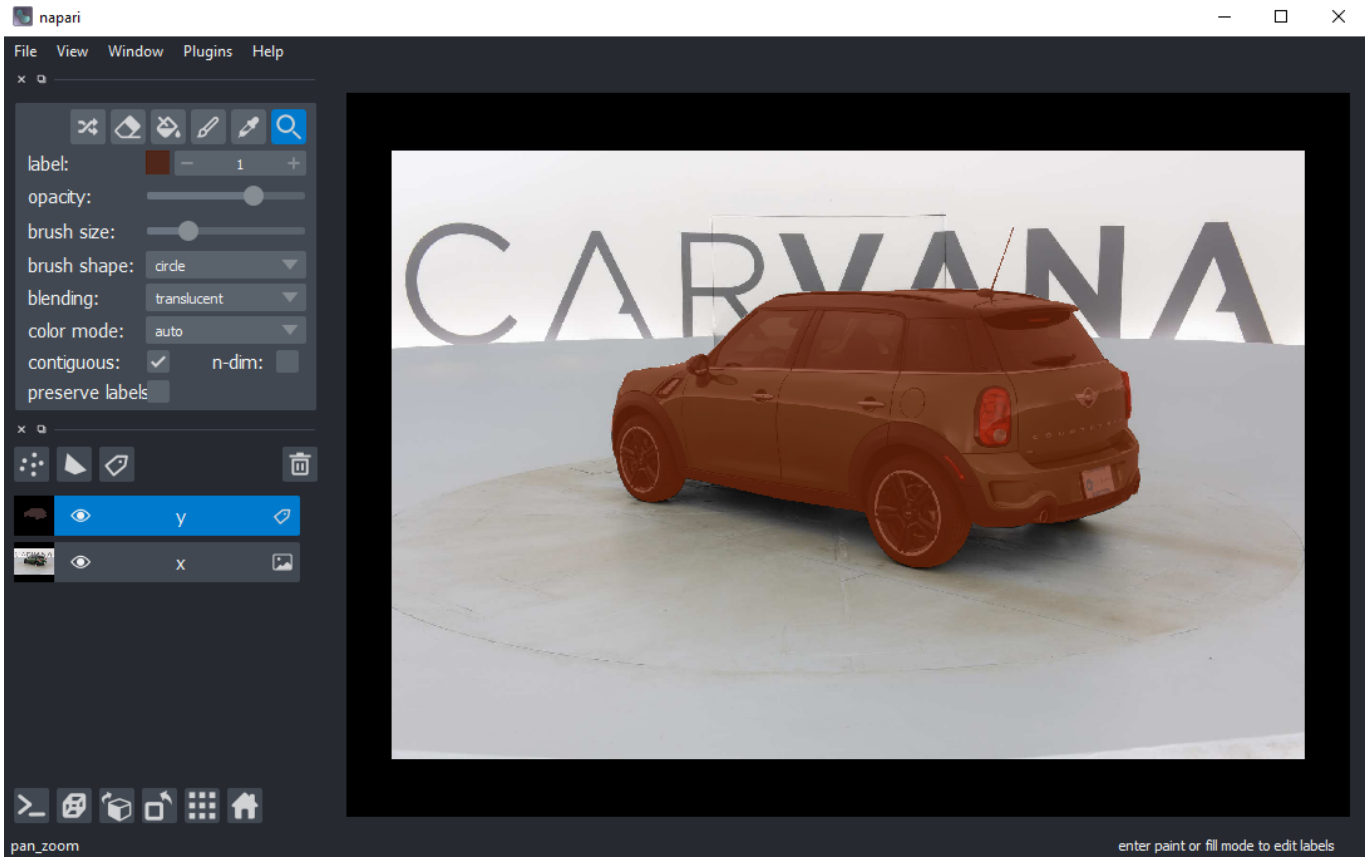

Image by Johannes Schmidt

model building (U-Net), training and inference. For that I will use a sample of the infamous Carvana dataset (2D images), but the code and the methods work for 3D datasets as well. For example: I will not use the torchvision package, as most transformations and augmentations only work on 2D input with PIL. You can find the repo of this project here with a jupyter notebook for every part (and more).

About myself

I am a master student in biology who recently (~2 years ago) started to learn programming with python and ended up trying deep learning (semantic segmentation with U-Net) on electron tomograms (3D images) for my master thesis. In this process I not only learned quite a lot about deep learning, python and programming but also how to better structure your project and code. This will be an attempt to share my experience and a tutorial to use plain PyTorch to efficiently use deep learning for your own semantic segmentation project.

Structure your project

To bring more structure into your project, I recommend to create a new project and create several python files, each having different classes and functions that can be imported and used. Personally, I like to use PyCharm for that but other IDEs such as Spyder are also a good choice. I also prefer working with the IPython console within PyCharm instead of Jupyter notebooks. A combination of both can be very helpful though. If you prefer a single Jupyter Notebook, that's also fine, just bear in mind that it could get to be a long script. You can find the files and code on github. Most of the structure and code that I will be showing is inspired by the work of this project: elektronn3.

About the data

Before we start creating our data generator, let's talk about the data first. What we would like to have is two directories, something like `/Input` and `/Target`. In the `/Input` directory, we find all input images and in the `/Target` directory the segmentation maps. Visualizing the images would look something like the image below. The labels are usually encoded with pixel values, meaning that all pixels of the same class have the same pixel value e.g. background=0, dog=1, cat=2 in the example below.

Image from chapter 13.9. Semantic Segmentation and the Dataset from the "Dive into Deep Learning" book — Semantically segmented image, with areas labeled 'dog', 'cat' and 'background — Creative Commons Attribution-ShareAlike 4.0 International Public License

The goal of the network is to predict such a segmentation map from a given input image.

Creating a dataset and a dataloader (data generator)

In deep learning, we want to feed our network with batches of data. Therefore, we would like to have a data generator that does the following:

1. It picks up the input-target pairs
2. It performs some transformations on the data "on the fly"

With PyTorch it is fairly easy to create such a data generator. We create a custom Dataset class, instantiate it and pass it to PyTorch's dataloader. Here is a simple example of such a dataset for a potential segmentation pipeline (Spoiler: In part 3 I will make use of the multiprocessing library and use caching to improve this dataset):

customdatasets.py

The `SegmentationDataSet` class inherits from `torch.data.Dataset`. In the initialization method `__init__`, we expect a list of input paths and a list of target paths. The `__getitem__` method simply reads an item from our input and target list using the `skimage.imread()` function. This will give us our input and target image as `numpy.ndarrays`. We then process our data with the `transform` function that expects an input and target pair and should return the processed data as `numpy.ndarrays` again. But I will come to that later. At the end, we just make sure that we end up having a

can be then used to create our dataloader (the data generator that we want). You may ask: How do we make sure that this dataset will output the correct target for every input image? If the input and target lists happen to have the same order, e.g. because input and target have the same name, the mapping should be correct.

Let's try it out with a simple example:

```
inputs = ['path\input\pic_01.png', 'path\input\pic_02.png']
targets = ['path\target\pic_01.png', 'path\target\pic_02.png']

training_dataset = SegmentationDataSet(inputs=inputs,
                                       targets=targets,
                                       transform=None)

training_dataloader = data.DataLoader(dataset=training_dataset,
                                      batch_size=2,
                                      shuffle=True)

x, y = next(iter(training_dataloader))

print(f'x = shape: {x.shape}; type: {x.dtype}')
print(f'x = min: {x.min()}; max: {x.max()}')
print(f'y = shape: {y.shape}; class: {y.unique()}; type: {y.dtype}')
```

Here, we create an instance of our `SegmentationDataSet` class. For now, we don't want any transformation to happen on our data, so we leave `transform=None` . In the next step we create our data loader by passing our `training_dataset` as input. We choose to have an `batch_size` of 2 and to shuffle the data `shuffle=True` . There are some other useful arguments that can be passed in when instantiating the dataloader and you should check them out. The result could look something like this:

```
x = shape: torch.Size([2, 144, 144, 3]); type: torch.float32
x = min: 8.0; max: 255.0
y = shape: torch.Size([2, 144, 144]); class: tensor([  0, 255]);
type: torch.int64
```

1. The type of x and y is already correct.
2. The x should have a shape of `[N, C, H, W]`. So the channel dimension should be second instead of last.
3. The y is supposed to have a shape `[N, H, W]`, (this is because torch's loss function `torch.nn.CrossEntropyLoss` expects it, even though it will internally one-hot encode it).
4. The target y has only 2 classes: 0 and 255. But we want dense integer encoding, meaning 0 and 1.
5. The input is in the range [0-255] — uint8, but should be normalized or linearly scaled to [0, 1] or [-1, 1].

Because of this we need to transform the data a little bit.

Note: If we omit the typecasting at the end of our `SegmentationDataSet`, the dataloader would still not output `numpy.ndarrays` but `torch.tensors`. As our input is read as a `numpy.ndarray` in `uint8`, it will also output a `torch.tensor` in `uint8`. For `float32` we need to change the type.

Adding transformations

Let's create a new file, that we name `transformations.py`.

In order to transform our data so that they meet the requirements of the network, we create a class `FunctionWrapperDouble` which can take any function and returns a partial (basically a function with defined arguments). This allows us to stack (functions) transformations in a list that we want to be applied on the data. To stack these functions I use the `ComposeDouble` class. The `Double` in these classes means that it is meant for a dataset with input-target pairs (like our case for semantic segmentation). There is also a `Single` version of these classes for the case that you want to just experiment with some input or target images (or visualize your dataset). So what we want, is to create an object, that comprises all the transformations that we want to apply to the data. We can

The `tranfsorms` variable is an instance of the `ComposeDouble` class that comprises a list of transformations that is to be applied to the data! `create_dense_target` and `normalize_01` are functions that I defined in transformations.py, while `np.moveaxis` is just a numpy function that simply moves the channel dimension C from last to second with the numpy library. It is important to note that these functions are expected to take in and output a `np.ndarray` ! This way, we can create and perform transformations including augmentations on arbitrary `numpy.ndarrays` . Or we could just write a wrapper class for other libraries, such as albumentations to make use of their transformations (more on that later). The `__repr__` is just a printable representation of a object and makes it easier to understand what transformations and what arguments were used.

Why the effort you may ask? Well, this makes it clear what transformations are performed on the data when we create the dataset. The `ComposeDouble` class just puts the different transformation together.

Let's test it out with some random input-target-data.

Here we also resize our input and target image by using the `skimage.transform.resize()` . And we linearly scale our input to the range [0, 1] using `normalize01` . We could also normalize the input based on a mean and std of a given dataset with `normalize` from the transformations.py file, but this will do for now.

This will output:

```
x = shape: (128, 128, 3); type: uint8
x = min: 0; max: 255
x_t: shape: (3, 64, 64)  type: float64
x_t = min: 0.0; max: 1.0
y = shape: (128, 128); class: [10 11 12 13 14]
y_t = shape: (64, 64); class: [0 1 2 3 4]
```

Building a dataloader for the Carvana dataset

`Carvana/Targets`. This is just a sample of the original dataset and can be downloaded here or found in the github repo. This sample consists of only 6 cars, but each car from 16 angles — 96 images in total. I will be using the `pathlib` library to get the directory path of every input and target. If this library is new to you, you should check it out here.

We import the `SegmentationDataSet` class and the transformations that we want to use. To get the input-target paths, I use the function `get_filenames_of_path` that will return a list of all items within a directory. We then stack our transformations inside the Compose object that we name `transforms`. Because training is usually performed with a training dataset and a validation dataset, we split our input and target lists with `sklearn.model_selection.train_test_split()`. We could do it manually too (and it makes more sense for this dataset), but that's ok for now. With these lists, we can create our datasets and the corresponding dataloaders. Now we should check if our datasets (training and validation) output the correct format! Since the datasets have a `__getitem__` and method, we can treat them almost like a sequence object (e.g. list):

This will give us:

```
x = shape: torch.Size([2, 3, 1280, 1918]); type: torch.float32
x = min: 0.0; max: 1.0
y = shape: torch.Size([2, 1280, 1918]); class: tensor([0, 1]); type:
torch.int64
```

To check if our dataloader functions the way we want, we can get a batch with:

```
x, y = next(iter(dataloader_training))
```

Everything seems to be in order now. Now let's visualize this result.

Visualizing the dataloader

however, it becomes a bit hacky and slow. But there is an alternative: napari. This is a fast, interactive multi-dimensional image viewer for python, that can be used in a jupyter notebook, an Ipython console or within a .py script.

> *It's built on top of `Qt` (for the GUI), `vispy` (for performant GPU-based rendering), and the scientific Python stack (`numpy`, `scipy`). — napari*

To visualize our dataset, we should reverse some of the transformations we performed on the data, e.g. we would like to have an input image of shape [H, W, C], in the range of [0–255] and as `numpy.ndarray`. Let's create a `visual.py` file in which we create a `DatasetViewer` class:

I won't go into details about napari and the code here. This class basically allows us to view our the images and targets of our dataset by iterating over the dataset with custom keybindings ('n' for next and 'b' for back)! Some comments on the transforms: The function `re_normalize()` scales the input image back to [0–255]. We assume that our images and targets are `torch.tensors` and force them to be on the cpu and as `np.ndarrays`.

Let's test it out:

You do not need to enter `%gui qt` before using napari in Ipython or witihn a Jupyter notebook in this case, because this will be evoked when intantiating the class with the `enabel_gui_qt()` function.

```
# open napari instance for training dataset
# navigate with 'n' for next and 'b' for back on the keyboard
# you can do the same for the validation dataset

from visual import DatasetViewer
dataset_viewer_training = DatasetViewer(dataset_train)

dataset_viewer_training.napari()
```

Image by Johannes Schmidt

When I press `'n'` on the keyboard, I will get the next image-target pair from the dataset:

Image by Johannes Schmidt

Now that we have a solid codebase, let's expand our dataset with some augmentations.

Adding augmentations

For 2D images, we don't have to implement everything from scratch with numpy. Instead, we could use a library like albumentations. For that, we just need to write a wrapper, like `AlbuSeg2d`. As an example, we could horizontally flip our training images and their respective targets. The validation images are usually not augmented, which makes it necessary to have different transformations for the training and validation dataset:

We can then change the code for our dataset objects accordingly:

The probability of a flipped image is `p=0.5`. When we visualize the image-target pairs again, we eventually get batches with horizontally flipped input-target pairs:

Summary

We have created a data generator that can feed a network with batches of transformed/augmented data in the correct format. We also know how to visualize our dataset with the help of napari. This applies to 2D and 3D datasets. Now that we have spent some time on creating and visualizing the dataset, let's move on to model building in the next chapter.