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 · 11 min read

In this series (4 parts) we will perform semantic segmentation on images using plain PyTorch and the U-Net architecture. I will cover the following topics: Dataset building, 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. A jupyter notebook of this part can be found here.

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 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

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 2 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.

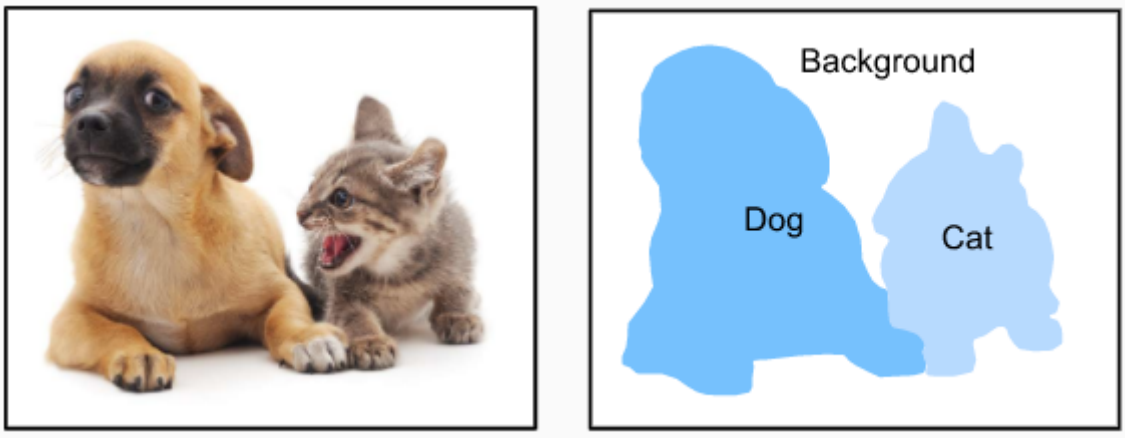

Image from chapter 13.9. Semantic Segmentation and the Dataset from the "Dive into Deep Learning" book — Semantically segmented image, with areas labeled 'dog', 'cat' and 'background — Creative Commons Attribution-ShareAlike 4.0 International Public License

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:

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):

```
import torch
from skimage.io import imread
from torch.utils import data


class SegmentationDataSet(data.Dataset):
    def __init__(self,
                 inputs: list,
                 targets: list,
                 transform=None
                 ):
        self.inputs = inputs
        self.targets = targets
        self.transform = transform
        self.inputs_dtype = torch.float32
        self.targets_dtype = torch.long

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self,
                    index: int):
        # Select the sample
        input_ID = self.inputs[index]
        target_ID = self.targets[index]

        # Load input and target
        x, y = imread(input_ID), imread(target_ID)

        # Preprocessing
        if self.transform is not None:
            x, y = self.transform(x, y)

        # Typecasting
```

customdatasets.py hosted with ❤️ by GitHub view raw

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 `torch.tensor` of certain type for our input and target. In this case, this is usually `torch.float32` and `torch.int64` (long) for input and target, respectively. This dataset 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}')
```

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
```

There are some things to notice here:

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

In order to transform our data so that they meet the requirements of the network, we create a class for every transformation we want to use. Why the effort? — Because I like having it clear and as comprehensible as possible. I want to immediately see what happens to the data once it is read by a dataloader. So what we want, is to create an object, that comprises all the transformations that we want to apply to the data. We can then pass this object as an argument in our `SegmentationDataSet` instance. This could look something like this:

```
transforms = Compose([
    DenseTarget(),
    MoveAxis(),
    Normalize01()
])
```

This makes it clear what transformations are performed on the data. The `Compose` class just puts the different transformation together. A transformation class has to have a `__call__` method, that takes in a single input and its respective target, processes it and outputs the processed input and target. As an example, take a look at the `MoveAxis` class. This simply moves the channel dimension C from last to second with the numpy library. 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.

Let's test it out.

```
x = np.random.randint(0, 256, size=(128, 128, 3), dtype=np.uint8)
y = np.random.randint(10, 15, size=(128, 128), dtype=np.uint8)

transforms = Compose([
    Resize(input_size=(64, 64, 3), target_size=(64, 64)),
    DenseTarget(),
    MoveAxis(),
    Normalize01()
```

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

print(f'y = shape: {y.shape}; class: {np.unique(y)}')
print(f'y_t = shape: {y_t.shape}; class: {np.unique(y_t)}')
```

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`, 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

Now let's put these pieces of code together to create a dataset for the Carvana dataset. We have our input images stored in `Carvana/Input` and our targets stored in `Carvana/Targets`. This is just a sample of the original dataset and can be downloaded here. 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.

```
from transformations import Compose, Resize, DenseTarget
from transformations import MoveAxis, Normalize01
from customdatasets import SegmentationDataSet
from torch.utils.data import DataLoader
```

```
# root directory
root = pathlib.Path('/Carvana')

def get_filenames_of_path(path: pathlib.Path, ext: str = '*'):
    """Returns a list of files in a directory/path. Uses pathlib."""
    filenames = [file for file in path.glob(ext) if file.is_file()]
    return filenames

# input and target files
inputs = get_filenames_of_path(root / 'Input')
targets = get_filenames_of_path(root / 'Target')

# training transformations and augmentations
transforms = Compose([
    DenseTarget(),
    MoveAxis(),
    Normalize01()
])

# random seed
random_seed = 42

# split dataset into training set and validation set
train_size = 0.8  # 80:20 split

inputs_train, inputs_valid = train_test_split(
    inputs,
    random_state=random_seed,
    train_size=train_size,
    shuffle=True)

targets_train, targets_valid = train_test_split(
    targets,
    random_state=random_seed,
    train_size=train_size,
    shuffle=True)

# dataset training
dataset_train = SegmentationDataSet(inputs=inputs_train,
                                    targets=targets_train,
                                    transform=transforms)

# dataset validation
dataset_valid = SegmentationDataSet(inputs=inputs_valid,
                                    targets=targets_valid,
                                    transform=transforms)

# dataloader training
dataloader_training = DataLoader(dataset=dataset_train,
                                 batch_size=2,
```

```
dataloader_validation = DataLoader(dataset=dataset_valid,
                                   batch_size=2,
                                   shuffle=True)
```

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 dataloader outputs the correct format every time we call the `__next__` function on our dataloader.

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

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}')
```

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
```

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

Visualizing the dataloader

In order to inspect the results of our dataloader, we should visualize the input and it's respective target image. For 2D images, we could use matplotlib. For 3D images, however, it becomes a bit hacky and slow. But there is an alternative: napari. This is a

> *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 batches, 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 class and a function.

gist.github.com/johschmidt42/e589047760ad224d97fde05ad3d2fad3.js

The class `Input_Target_Pair_Generator` is another generator that randomly picks one image-target pair from a generated batch and transforms it into the desired output. For example, it uses the function `re_normalize()` to scale the input image back to [0–255]. This generator can then be used to create a napari viewer instance. For this, I create the function `show_input_target_pair_napari()`, that takes in such a generator and visualizes a input-target pair. This function could also be placed within the class.

Let's test it out:

Please do not forget to enter `%gui qt` before using napari in Ipython or witihn a Jupyter notebook.

```
from visual import Input_Target_Pair_Generator
from visual import show_input_target_pair_napari

gen = Input_Target_Pair_Generator(dataloader_training, rgb=True)
show_input_target_pair_napari(gen)
```

This will open a GUI that will look something like this:

Image by Johannes Schmidt

When I press `'t'` on the keyboard, I will get the next random image-target pair from the next batch:

Image by Johannes Schmidt

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:

```
# training transformations and augmentations
transforms_training = Compose([
    # Resize(input_size=(128, 128, 3), target_size=(128, 128)),
    AlbuSeg2d(albu=albumentations.HorizontalFlip(p=0.5)),
    DenseTarget(),
    MoveAxis(),
    Normalize01()
])

# validation transformations
transforms_validation = Compose([
    # Resize(input_size=(128, 128, 3), target_size=(128, 128)),
    DenseTarget(),
    MoveAxis(),
    Normalize01()
])
```

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

```
# dataset training
dataset_train = SegmentationDataSet(inputs=inputs_train,
                                    targets=targets_train,
                                    transform=transforms_training)

# dataset validation
dataset_valid = SegmentationDataSet(inputs=inputs_valid,
                                    targets=targets_valid,
                                    transform=transforms_validation)
```

Image by Johannes Schmidt

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 these batches of data 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.

Unet Deep Learning Pytorch Semantic Segmentation Python