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

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

Johannes Schmidt Jan 18 · 6 min read

In the previous chapters we built a dataloader, created a configurable U-Net model, and started training with a sample of the Carvana dataset. Now let's use this trained model to predict the segmentation maps of some images that have not been seen by the network. You can find three images of the Carvana dataset here. The Jupyter notebook for this part can be found here and the GitHub repo here.

Inference

For inference we need to perform some transformations on the data before we pass it through the network. This includes linear scaling, changing the order of the dimensions (we want [C, H, W] instead of [H, W, C]) and resizing the image. Therefore, I could write a simple `predict()` function that takes in the image as `np.ndarray`, the model, a preprocess and a postprocess function, and a device where inference should be performed on. The input image is preprocessed (line 11), transformed to a `torch.tensor` and sent to a device (line 12). Then it is sent through the model which outputs the logits (line 14). Please note that no Softmax activation function is applied at the end of the UNet model, therefore one has to add it for inference (line 16). The result should be

```python
import torch


def predict(img,
            model,
            preprocess,
            postprocess,
            device,
            ):
    model.eval()
    img = preprocess(img)  # preprocess image
    x = torch.from_numpy(img).to(device)  # to torch, send to device
    with torch.no_grad():
        out = model(x)  # send through model/network

    out_softmax = torch.softmax(out, dim=1)  # perform softmax on outputs
    result = postprocess(out_softmax)  # postprocess outputs

    return result
```

inference.py hosted with ♡ by GitHub view raw

Let's load our trained model first

```
import torch
from unet import UNet

# device
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    torch.device('cpu')

# model
model = UNet(in_channels=3,
             out_channels=2,
             n_blocks=4,
             start_filters=32,
             activation='relu',
             normalization='batch',
             conv_mode='same',
             dim=2).to(device)
```

```
model_weights = torch.load(pathlib.Path.cwd() / model_name)

model.load_state_dict(model_weights)
```

and use it for inference:

```
# Imports
import pathlib

import numpy as np
import torch
from skimage.io import imread
from skimage.transform import resize

from inference import predict
from transformations import normalize_01, re_normalize
from unet import UNet

# root directory
root = pathlib.Path.cwd() / 'Carvana' / 'Test'
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
images_names = get_filenames_of_path(root / 'Input')
targets_names = get_filenames_of_path(root / 'Target')

# read images and store them in memory
images = [imread(img_name) for img_name in images_names]
targets = [imread(tar_name) for tar_name in targets_names]

# Resize images and targets
images_res = [resize(img, (128, 128, 3)) for img in images]
resize_kwargs = {'order': 0, 'anti_aliasing': False, 'preserve_range': True}
targets_res = [resize(tar, (128, 128), **resize_kwargs) for tar in targets]

# device
if torch.cuda.is_available():
    device = torch.device('cuda')
```

```python
# model
model = UNet(in_channels=3,
             out_channels=2,
             n_blocks=4,
             start_filters=32,
             activation='relu',
             normalization='batch',
             conv_mode='same',
             dim=2).to(device)


model_name = 'carvana_model.pt'
model_weights = torch.load(pathlib.Path.cwd() / model_name)

model.load_state_dict(model_weights)

# preprocess function
def preprocess(img: np.ndarray):
    img = np.moveaxis(img, -1, 0)  # from [H, W, C] to [C, H, W]
    img = normalize_01(img)  # linear scaling to range [0-1]
    img = np.expand_dims(img, axis=0)  # add batch dimension [B, C, H, W]
    img = img.astype(np.float32)  # typecasting to float32
    return img


# postprocess function
def postprocess(img: torch.tensor):
    img = torch.argmax(img, dim=1)  # perform argmax to generate 1 channel
    img = img.cpu().numpy()  # send to cpu and transform to numpy.ndarray
    img = np.squeeze(img)  # remove batch dim and channel dim -> [H, W]
    img = re_normalize(img)  # scale it to the range [0-255]
    return img

# predict the segmentation maps
output = [predict(img, model, preprocess, postprocess, device) for img in images_res]
```

inference_example.py hosted with ♡ by GitHub view raw

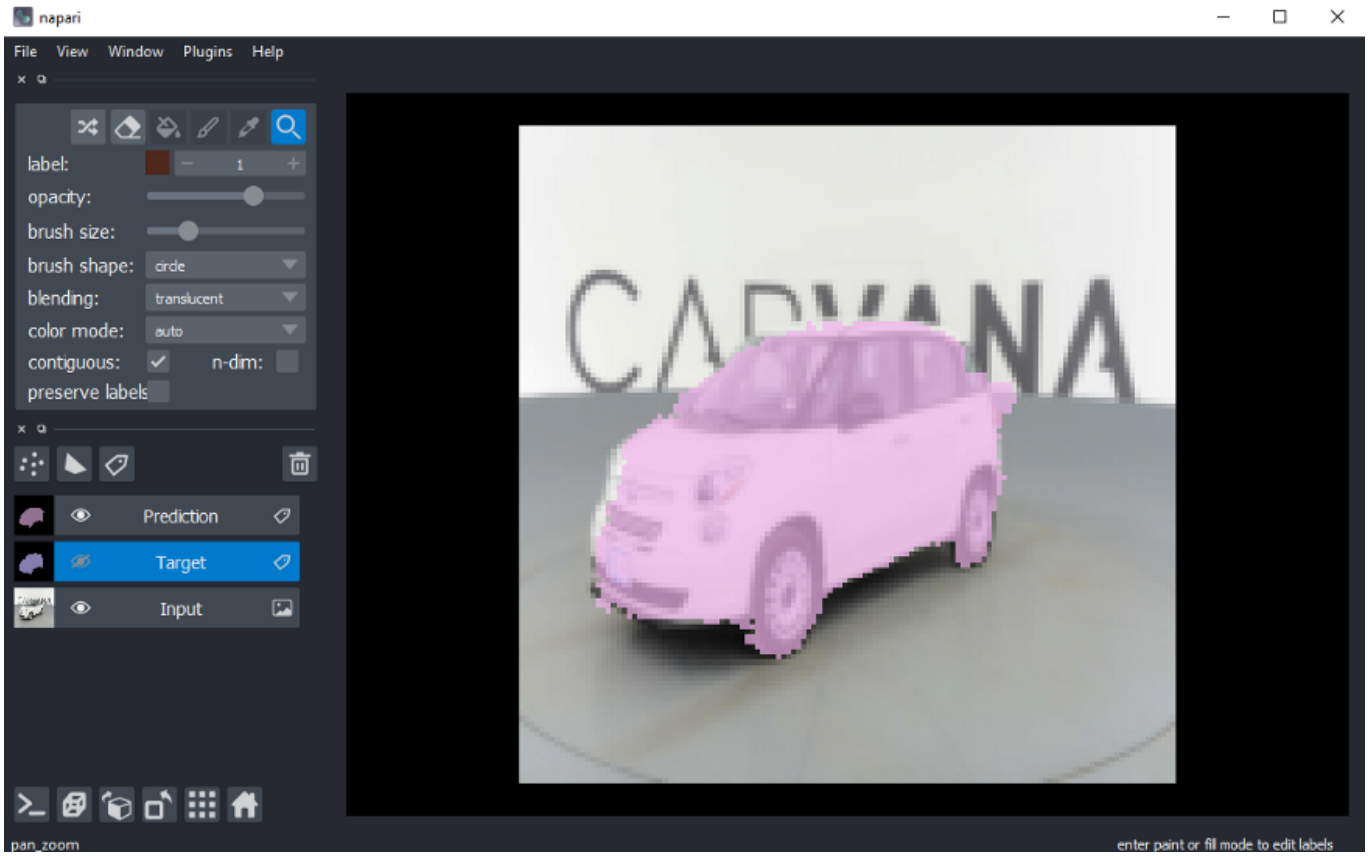
The input and target paths that are obtained by `get_filenames_of_path` are read with `skimage.io.imread()` and stored in a list: `images` and `targets`. Since training was

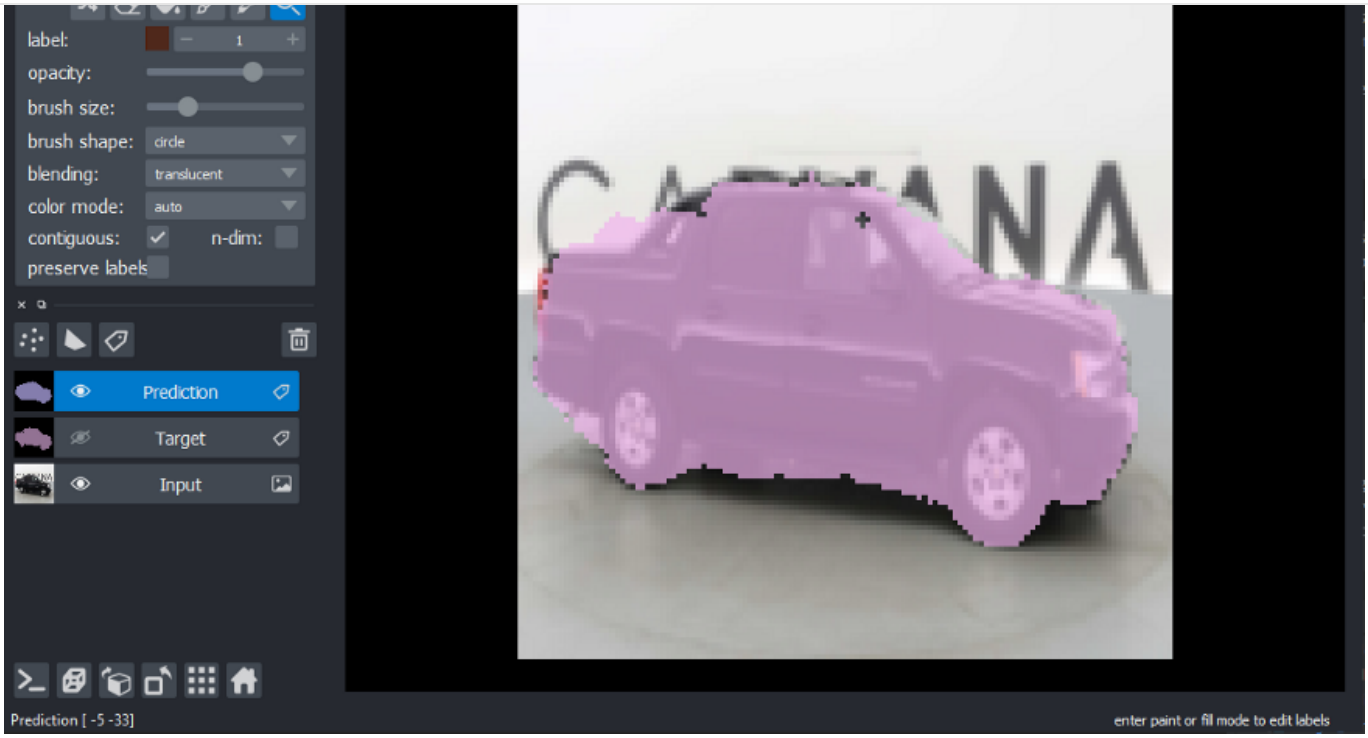
resized to 128x128x3. In the preprocess function we just perform the same steps we have performed on the training/validation data but without augmentations. In the postprocess function we perform `argmax` on our output, because we don't want the probabilities of every class (softmaxed logits), but instead the predicted class, which is the class with the highest probability. We sequentially use the model for inference to process our images and store the result in the list `outputs`. We can then use napari again to visualize the outcome and compare it to its actual ground-truth.

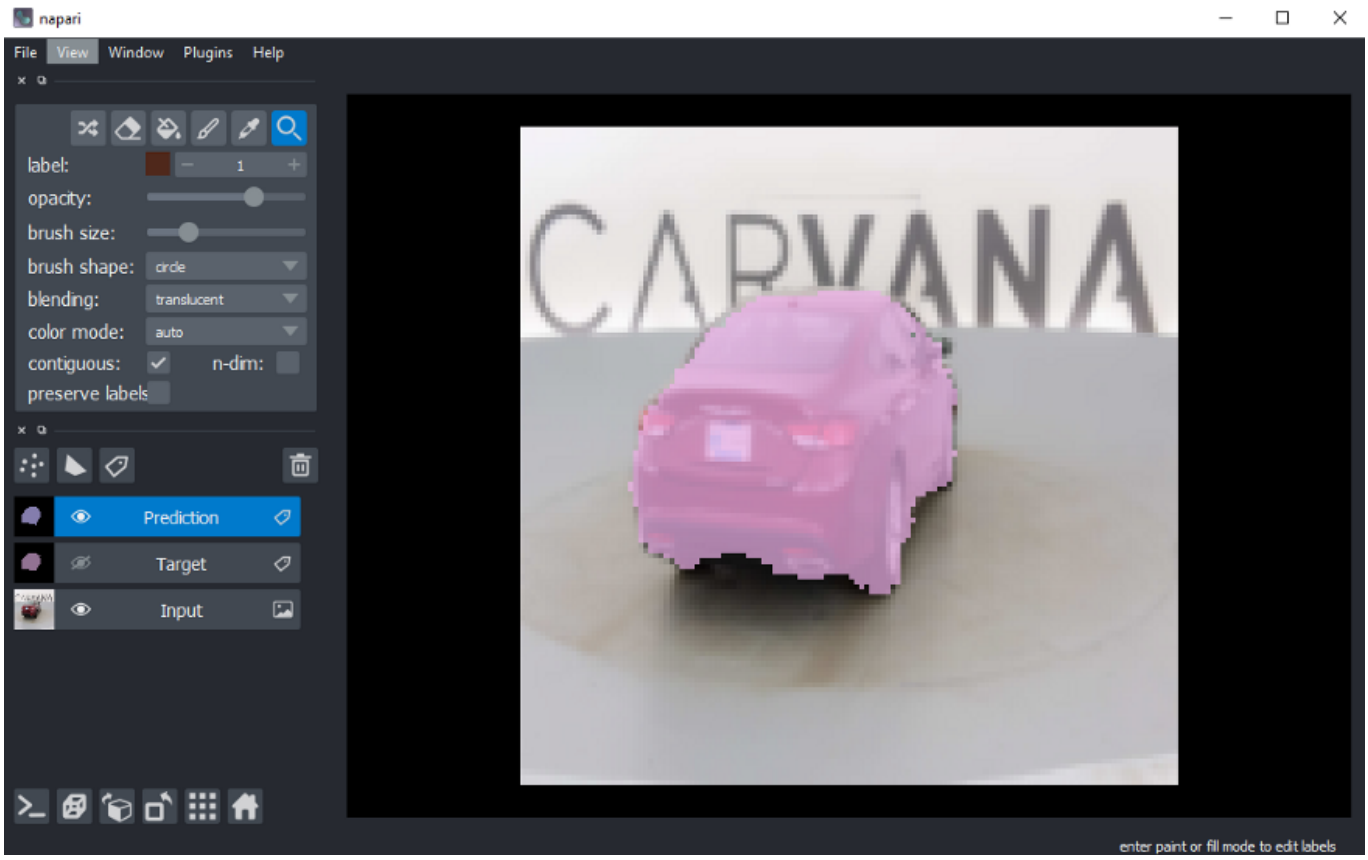
```
import napari

viewer = napari.Viewer()

idx = 0
img_nap = viewer.add_image(images_res[idx], name='Input')
tar_nap = viewer.add_labels(targets_res[idx], name='Target')
out_nap = viewer.add_labels(output[idx], name='Prediction')
```







The predictions are not too bad, but not good either. So there's a lot of room for improvement! But what can you do to improve your model? These are some tips that

- Training should be performed on all the available data, not just the 90 images I picked from the dataset.
- There are 16 images per car which all should either be present in the training OR in the validation dataset, not both! Remember that the purpose of the validation dataset is to determine the time to stop training. Implement early stopping to find the right time.
- Increase the resolution of the images. Generally speaking, more features can be extracted from higher resolution images.
- Add more augmentations, not just horizontal flips. This artificially extends the size of your training dataset which might make your model more robust and generalize better.
- Experiment with hyperparameters! Try a different optimizer like Adam instead of SGD, use different activation functions, increase the number of kernels, use different normalization layers, different upsampling methods etc.
- Try out the dice loss instead of standard CrossEntropyLoss. Or use a combination of both!
- Consider using transfer learning to learn the task faster. Replace the UNet with one of the segmentation models found here. These models (including UNet) can have different backbones and are pretrained on e.g. ImageNet.
- Use an evaluation metric such as Intersection over Union (IoU) to measure the accuracy of the model on a particular dataset. Besides visualizing the results, this will give you a better feeling on how good your model is.
- Try out mixed precision learning to be able to increase your batch size or consider using gradient accumulation when dealing with high resolution images that fill up the memory very quickly.

Summary

In this last part, we used our trained model to make predictions (segmentation maps) on images that have not been seen by the network. Inference is basically just passing the

room for improvement.

This tutorial is supposed to serve as a starting point and an example for a semantic segmentation project with deep learning. I recommend you go to Kaggle and download the complete dataset and see for yourself how well you could train your model. You could also start your own semantic segmentation project with either an existing dataset like Cityscapes (2D), the 2019 Kidney Tumor Segmentation Challenge (KiTS19, 3D), etc. or create your own dataset, for which you have to provide labels. And as you have seen in this tutorial, training can be carried out very easily in plain PyTorch. But you should actually do it only if you want to learn PyTorch. It is generally recommended to use higher level APIs, such as Lightning, Fast.ai or Skorch. But why? This is well explained in this article.

You can probably imagine that if you want to integrate functionalities and features, such as logging, metrics, early stopping, mixed precision training and many more to your training loop, you'll end up doing exactly what others have done already. However, chances are that your code won't be as good and stable as theirs (hello spagetthi code) and you'll spent too much time on integrating and debugging these things rather than focusing on your deep learning project (hello me). And although learning a new API can take some time, it might help you a lot in the long run.

If you want to implement your own features anyway, take a look at this article, and use callbacks and hooks. For this series, I tried to keep it very simple and without the use of out-of-the-box solutions. I have implemented a very basic data processing pipeline/training loop with good ol' PyTorch that works for both 2D and 3D datasets.

Alright, that's it for this little tutorial. I hope you enjoyed it, learnt something new, and hope I could make it easier for you to start your own deep learning segmentation project. There are plenty of useful tutorials and projects in the vastness of the internet. Thank you for reading!

Pytorch Python Semantic Segmentation Unet Deep Learning