Open in app

Follow 579K Followers

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

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

Johannes Schmidt Dec 5, 2020 · 8 min read

In the previous chapters we created our dataset and built the U-Net model. Now it is time to start training. For that we will write our own training loop within a simple `Trainer` class and save it in `trainer.py`. The Jupyter notebook can be found here. The idea is that we can instantiate a `Trainer` object with parameters such as the `model`, a `criterion` etc. and then call it's class method `run_trainer()` to start training. This method will output the accumulated training loss, the validation loss, and the learning rate that was used for training. Here is the code:

```
import numpy as np
import torch


class Trainer:
    def __init__(self,
                 model: torch.nn.Module,
                 device: torch.device,
                 criterion: torch.nn.Module,
                 optimizer: torch.optim.Optimizer,
                 training_DataLoader: torch.utils.data.Dataset,
                 validation_DataLoader: torch.utils.data.Dataset = None
```

Open in app

```
                 epoch: int = 0,
                 notebook: bool = False
                 ):

        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.lr_scheduler = lr_scheduler
        self.training_DataLoader = training_DataLoader
        self.validation_DataLoader = validation_DataLoader
        self.device = device
        self.epochs = epochs
        self.epoch = epoch
        self.notebook = notebook

        self.training_loss = []
        self.validation_loss = []
        self.learning_rate = []

    def run_trainer(self):

        if self.notebook:
            from tqdm.notebook import tqdm, trange
        else:
            from tqdm import tqdm, trange

        progressbar = trange(self.epochs, desc='Progress')
        for i in progressbar:
            """Epoch counter"""
            self.epoch += 1  # epoch counter

            """Training block"""
            self._train()

            """Validation block"""
            if self.validation_DataLoader is not None:
                self._validate()

            """Learning rate scheduler block"""
            if self.lr_scheduler is not None:
                if self.validation_DataLoader is not None and self.lr_scheduler.__class
                    self.lr_scheduler.batch(self.validation_loss[i])  # learning rate s
```

Open in app

```

    def _train(self):

        if self.notebook:
            from tqdm.notebook import tqdm, trange
        else:
            from tqdm import tqdm, trange

        self.model.train()  # train mode
        train_losses = []  # accumulate the losses here
        batch_iter = tqdm(enumerate(self.training_DataLoader), 'Training', total=len(se
                          leave=False)

        for i, (x, y) in batch_iter:
            input, target = x.to(self.device), y.to(self.device)  # send to device (GPU
            self.optimizer.zero_grad()  # zerograd the parameters
            out = self.model(input)  # one forward pass
            loss = self.criterion(out, target)  # calculate loss
            loss_value = loss.item()
            train_losses.append(loss_value)
            loss.backward()  # one backward pass
            self.optimizer.step()  # update the parameters

            batch_iter.set_description(f'Training: (loss {loss_value:.4f})')  # update

        self.training_loss.append(np.mean(train_losses))
        self.learning_rate.append(self.optimizer.param_groups[0]['lr'])

        batch_iter.close()

    def _validate(self):

        if self.notebook:
            from tqdm.notebook import tqdm, trange
        else:
            from tqdm import tqdm, trange

        self.model.eval()  # evaluation mode
        valid_losses = []  # accumulate the losses here
        batch_iter = tqdm(enumerate(self.validation_DataLoader), 'Validation', total=le
                          leave=False)

```

Open in app

```
105                 with torch.no_grad():
106                     out = self.model(input)
107                     loss = self.criterion(out, target)
108                     loss_value = loss.item()
109                     valid_losses.append(loss_value)
110 
111                     batch_iter.set_description(f'Validation: (loss {loss_value:.4f})')
112 
113             self.validation_loss.append(np.mean(valid_losses))
114 
115             batch_iter.close()
```

In order to create a trainer object the following parameters are required:

- `model` : e.g. the U-Net
- `device` : CPU or GPU
- `criterion` : loss function (e.g. CrossEntropyLoss, DiceCoefficientLoss)
- `optimizer` : e.g. SGD
- `training_DataLoader` : a training dataloader
- `validation_DataLoader` : a validation dataloader
- `lr_scheduler` : a learning rate scheduler (optional)
- `epochs` : The number of epochs we want to train
- `epoch` : The epoch number from where training should start

Training can then be started with the class method `run_trainer()` . Since training is usually performed with a training and a validation phase, `_train()` and `_validate()` are two functions that are run once for every epoch we train with `run_trainer()` (line 33–53). If we have a lr_scheduler, we also perform a step with the lr_scheduler. To visualize the progress of training, I included a progress bar with the library tqdm. Now let's take a

Open in app

In `_train()` we basically just iterate over our training dataloader and send our batches through the network in train mode (line 56–64). We then use this output together with our target to compute the loss with the loss function for the current batch (line 65). The computed loss is then appended in a temporary list (line 66–67). Based on the computed gradients, we perform a backward pass and a step with our optimizer to update the model's parameters (line 68–69). At the end we update our progress bar for the training phase to show the current loss (line 71). The function outputs the mean of the temporary loss list and the learning rate that was used.

In `_validate()` , similar to `_train()` , we iterate over our validation dataloader, send our batches through the network in validation mode and compute the loss. This time, without computing the gradients and without performing a backward pass (line 78–97).

Start training with the Carvana dataset

Let's create our Carvana data generators once again, but this time run the code within a Jupyter notebook.

```
# Imports
import pathlib
from transformations import Compose, AlbuSeg2d, DenseTarget
from transformations import MoveAxis, Normalize01, Resize
from sklearn.model_selection import train_test_split
from customdatasets import SegmentationDataSet
import torch
from unet import UNet
from trainer import Trainer
from torch.utils.data import DataLoader
import albumentations

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

Open in app

```
    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()
])

# 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)

# inputs_train, inputs_valid = inputs[:80], inputs[80:]
# targets_train, targets_valid = targets[:80], targets[:80]

# 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)

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

Open in app

```
shuffle=True)
```

Please note that I resize the images to 128x128x3 using `Resize()` to speed up training. This will generate batches of images that look like this:

```
%gui qt
from visual import Input_Target_Pair_Generator
from visual import show_input_target_pair_napari

gen_t = Input_Target_Pair_Generator(dataloader_training, rgb=True)
gen_v = Input_Target_Pair_Generator(dataloader_validation, rgb=True)
show_input_target_pair_napari(gen_t, gen_v)
```

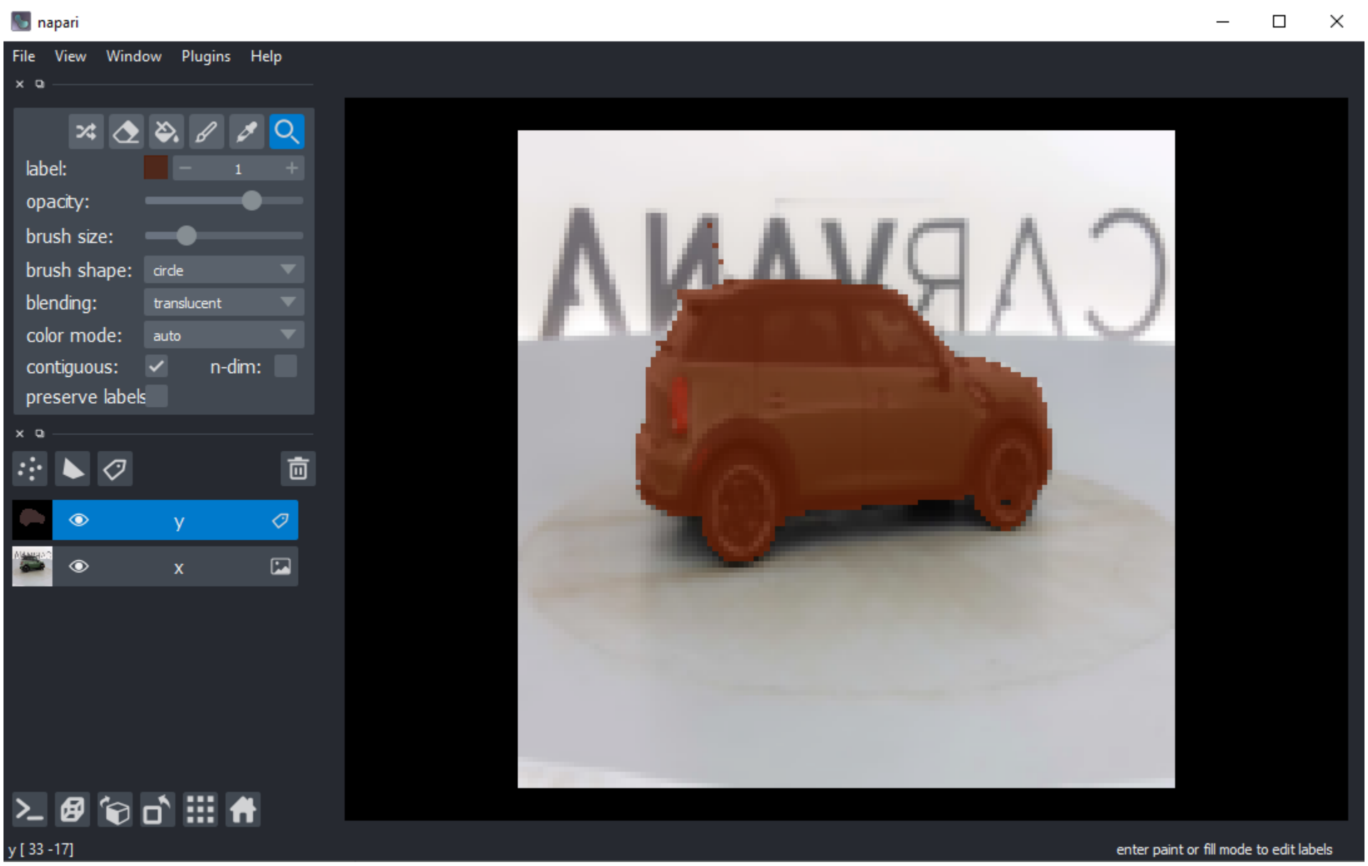

Image by author

I can then instantiate the `Trainer` object and start training:

Open in app

```
    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)

# criterion
criterion = torch.nn.CrossEntropyLoss()

# optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# trainer
trainer = Trainer(model=model,
                  device=device,
                  criterion=criterion,
                  optimizer=optimizer,
                  training_DataLoader=dataloader_training,
                  validation_DataLoader=dataloader_validation,
                  lr_scheduler=None,
                  epochs=10,
                  epoch=0,
                  notebook=True)

# start training
training_losses, validation_losses, lr_rates = trainer.run_trainer()
```

Training will look something like this:

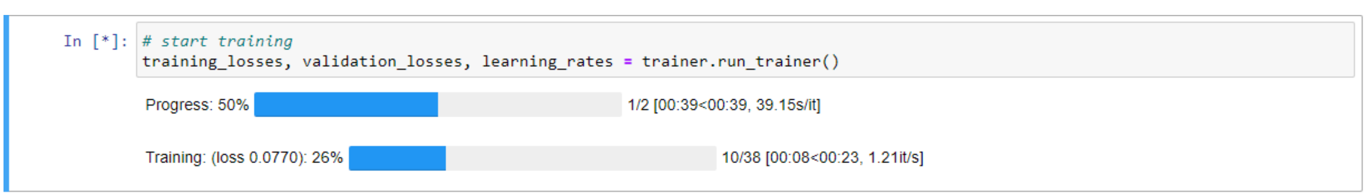

Image by author

Improve the data generator

Open in app

is so painfully slow, is because every time we generate a batch we read the data in full resolution (1918x1280x3) and resize it. And we do this for every epoch! Therefore, it would make more sense to either store the data in a lower resolution and then to pick the data up, or store the data in cache and access it when it's needed. Or both. Let's slightly change our custom `SegmentationDataSet` class (I will create a new file and name it `customdatasets2.py`, but you can replace your `customdatasets.py` with this one):

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


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

        if self.use_cache:
            self.cached_data = []

            progressbar = tqdm(range(len(self.inputs)), desc='Caching')
            for i, img_name, tar_name in zip(progressbar, self.inputs, self.targets):
                img, tar = imread(img_name), imread(tar_name)
                if self.pre_transform is not None:
                    img, tar = self.pre_transform(img, tar)

                self.cached_data.append((img, tar))

```

Open in app

```
37      def __getitem__(self,
38                      index: int):
39          if self.use_cache:
40              x, y = self.cached_data[index]
41          else:
42              # Select the sample
43              input_ID = self.inputs[index]
44              target_ID = self.targets[index]
45
46              # Load input and target
47              x, y = imread(input_ID), imread(target_ID)
48
49          # Preprocessing
50          if self.transform is not None:
51              x, y = self.transform(x, y)
52
53          # Typecasting
54          x, y = torch.from_numpy(x).type(self.inputs_dtype), torch.from_numpy(y).type(sel
55
56          return x, y
```

customdatasets2.py hosted with ❤ by GitHub view raw

Here we added the argument `use_cache` and `pre_transform` . We basically just iterate over our input and target list and store the images in a list when we instantiate our dataset. When `__getitem__` is called, an image-target pair from this list is returned. I added the `pre_transform` argument because I don't want to change the original files. Instead, I want the images to be picked up, resized and stored in memory. Again, I included a progress bar to visualize the caching. If you want to run it correctly in Jupyter, please import `tqdm` from `tqdm.notebook` in line 4 in `customdatasets2.py` . Let's try it out. The changes in code are the following:

```
# pre-transformations
pre_transforms = Compose([
    Resize(input_size=(128, 128, 3), target_size=(128, 128)),
])

# training transformations and augmentations
transforms_training = Compose([
```

Open in app

```
    Normalize01()
])

# validation transformations
transforms_validation = 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)
```

And it looks something like this:

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

# dataloader validation
dataloader_validation = DataLoader(dataset=dataset_valid,
                                   batch_size=2,
                                   shuffle=True)

Caching: 100%|██████████| 76/76 [00:26<00:00,  2.83it/s]
Caching:   5%|▌         | 1/20 [00:00<00:07,  2.47it/s]
```

Image by author

The first progress bar represents the training dataloader and the second the validation dataloader. Let's train again for 2 epochs and see how long it'll take.

Open in app

Image by author

Training took about 2 seconds only! That's much better. But there is one part we can still improve. Creating the dataset that reads images and stores them in memory takes a bit of time. When you look at the code and the CPU usage, you'll notice that only one core is used. Let's change it in a way, so that all cores are used. Here I use the multiprocessing library.

```
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,
                 use_cache=False,
                 pre_transform=None,
                 ):
        self.inputs = inputs
        self.targets = targets
        self.transform = transform
        self.inputs_dtype = torch.float32
        self.targets_dtype = torch.long
        self.use_cache = use_cache
        self.pre_transform = pre_transform

        if self.use_cache:
            from multiprocessing import Pool
            from itertools import repeat

            with Pool() as pool:
                self.cached_data = pool.starmap(self.read_images, zip(inputs, targets, r

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

Open in app

```
        if self.use_cache:
            x, y = self.cached_data[index]
        else:
            # 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
        x, y = torch.from_numpy(x).type(self.inputs_dtype), torch.from_numpy(y).type(sel

        return x, y

    @staticmethod
    def read_images(inp, tar, pre_transform):
        inp, tar = imread(inp), imread(tar)
        if pre_transform:
            inp, tar = pre_transform(inp, tar)
        return inp, tar
```

Before we perform training, let's also make a quick detour and talk about the learning rate.

Learning rate finder

The learning rate is one of the most important hyperparameters in neural network training. Choosing proper learning rates throughout the learning procedure is difficult as a small learning rate leads to slow convergence while a high learning rate can cause divergence. Also, frequent parameter updates with high variance in SGD can cause fluctuations, which makes finding the (local) minimum for SGD even more difficult. To identify an optimal learning rate, we can test different learning rates empirically with a learning rate range test. Inspired by the best practices I picked up from the fast.ai course,

Open in app

will show you is based on Tanjid Hasan Tonmoy's pytorch-lr-finder, which is an implementation of the learning rate range test from Leslie Smith. I only slightly modified the code and included a progressbar (yes, I like them).

Let's perform such a learning rate range test. Since our dataset is rather small (96 images), we'll perform some extra steps (1000). The upper progressbar displays the number of epochs and the lower progressbar shows the number of steps we perform on the current epoch.

Image by author

Let's plot the results of the test:

Image by author

0.01 seems to be a good learning rate. We'll take it. Let's train for 100 epochs…

Open in app

Image by author

…and visualize the training and validation loss. For that I will use matplotlib and write a function that I can add to the `visual.py` file.

Let's see what the function `plot_training()` will output when we pass in our losses and the learning rate.

Image by author

Training looks good!

We can then save our model with PyTorch:

Open in app

```
torch.save(model.state_dict(), pathlib.Path.cwd() / model_name)
```

Summary

In this part, we performed training with a sample of the Carvana dataset by creating a simple training loop. The progress of this training loop can be visualized with a progressbar and the result of training can be plotted with matplotlib. We noticed that training was painfully slow because our data was picked up very slowly by our custom data generator. Because of that, we changed it in a way so that data is only read once and then picked up from memory when needed. We also made use of multiprocessing for that case. Additionally, we added a learning rate range finder, to determine an optimal learning rate which we then used for model training.

In the next chapter, we'll let the model predict the segmentation maps of unseen image data.

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

Emails will be sent to mr.spons@gmail.com.
Not you?

Python Deep Learning Semantic Segmentation Pytorch Tutorial

About Help Legal

Get the Medium app

Open in app