Exercise 9: Semantic Segmentation with Deep Learning

Introduction

In the previous exercise, we implemented a full routine for training and validating DL models for image classification. Now, we shall take it to the next level and perform pixel-wise classification, also known as semantic segmentation. This is of particular interest to remote sensing, as it allows us to *e.g.* obtain spatially well-resolved land cover maps, among other products.
The basic ingredients are exactly the same as for image classification (optimiser, loss function, training loop, etc.). We do have some changes to make, though:

- Dataset: this time, we do not just want a single output number (class), but one value per spatial location. Essentially, our dataset should also provide a second image where each pixel has the class index as its value.
- Model: likewise, we need a suitable model that provides spatial outputs rather than a single class vector. You have seen some examples in the lecture. In this exercise we shall use a flavour of the Hypercolumn to do this job.

Info

- All parts in the code that require your input for completion are marked with flag "`#TODO`".
- PyTorch has a very elaborate online documentation:
  `https://pytorch.org/docs/stable/index.html`.
  You are encouraged to consult it frequently!

Tasks

1 Setup

1.1 Prepare your environment: you will need a GPU for this exercise. You can either use Colab, or connect to the EPFL's cluster.
1.2 In the Web browser window, open the Jupyter Notebook `ex8.ipynb`.
1.3 Install the required dependencies by running the first code block.

1.4 Check if the Graphics Processing Unit (GPU) is available within your environment and PyTorch. Do so by running the respective code cell.
1.5 Also, set the random seed again for reproducibility.

2 Dataset

For this exercise we shall be using the ISPRS Vaihingen semantic segmentation dataset. This is a set of fully-labelled satellite image-segmentation mask pairs, with 9 cm resolution and six land cover classes: Impervious, Buildings, Low Vegetation, Tree, Car, Clutter. The images come from a large satellite scene over the town Vaihingen in Germany and were divided into 33 patches, some of which are available with ground truth. These patches are still too large for our model: we would quickly run out of GPU memory if we tried to process an image of *e.g.* 4000×3000 pixels. Hence, they need to be further divided into even smaller patches. This has already been done for you; all you need to do is to download the image-label pair patches (sized 512×512 pixels) by running the code cell below.

2.1 Download the prepared ISPRS Vaihingen dataset patches (code provided).
2.2 You are provided with a new PyTorch Dataset class for this dataset that does all data organisation, loading and preparing for you. The respective code block also provides you with a new function for setting up a PyTorch DataLoader on this dataset.
2.3 Run the next provided code block to visualise images and segmentation masks (labels). As you can see, we do not just have visual imagery this time, but also a height model (a digital surface model). So we need to create a model that can accept four input bands accordingly (infrared, red, green, DSM).

3 Model

As explained above, we shall be using a Hypercolumn for the task of semantic segmentation. Hypercolumn is just one of multiple models that can perform semantic segmentation, and there are more complex (and oftentimes more accurate) models these days. However, Hypercolumn has a very elegant working principle and still performs reasonably well, as you will see.

3.1 Read the explanation of Hypercolumns in the Notebook and try to understand how it works.

`CODE` Implement your own Hypercolumn according to the blueprint provided in the description. Like last week, this requires you to implement an object class (`class Hypercolumn(nn.Module)`), including its constructor and forward pass. Then, run the subsequent code block to test whether your model provides the right output.

4 Model training

Like last week, we now have a dataset and a model to match the objective. So let's put the pieces together and train it! The good news here is that you can literally copy-paste the vast majority of code you developed last week! Both tasks were about classification, the main difference in objective is that we now do pixel-wise classification. The pieces in PyTorch (loss,*etc.*) can cope with both the same way, though.

4.1 Implement the loss function, optimiser, training epoch and validation epoch functions.

`CODE` Provide all the code blocks until the one that is provided again by copy-pasting them from the previous exercise.

4.2 Train your model by running all code blocks in Section 4.

5 Model validation

Above we assessed the model's accuracy during training on the held-out validation set. This time, we unfortunately do not have a test set, since the ISPRS Vaihingen dataset was originally designed as a contest, where people would compete against each other! Hence, the labels for the test set are still hidden and only accessible via the public evaluation server.
However, we can do something else here that we couldn't before: visualise our predictions!

5.1 Visualise the predictions on five random images from the validation set by running the provided code block. By default, this will load the model at epochs zero (fresh initialisation), one, five, and the last saved checkpoint.

Q Can you see a pattern in how the model evolves along the training epochs? If so, can you explain it (*e.g.*, sudden jumps in the quality of the prediction result)?