UNIVERSITY OF CAMBRIDGE

Data Science in Medical Imaging

MPhil in Data Intensive Science

Monday 26th February 2024 (3:00 pm to 5:00 pm)

Minor 2: Data Science Applications to Medical Imaging
- Problem Sheet 3: Image Detection and Segmentation
Dr L. Escudero Sánchez

In this practical session you will gain experience in basic concepts of image detection as well as in writing from scratch a simple 2D UNet for image segmentation.

*Attempt **all** questions.*

We will work through the questions during the practical session on the date indicated above.

Solutions to these questions will be available on Moodle after such practical session.

[Prep work before the session]
Get the skeleton Jupyter notebook called Practical3_empty.ipynb from https://github.com/loressa/DataScience_MPhill_practicals.git (Practical3). Download as well the pre-trained model (SimpleUNet_v3.pt) and the test numpy file (Case_010.npz).
For the programming part of this session we will use `PyTorch` and associated libraries.
You have two options:

- Work in Google Colaboratory (recommended)
- Install the approrpiate libraries locally (wherever you will be running the exercise)

In either case, the packages and versions needed are the following:

- `torch` version `2.1.0+cu121`
- `torchmetrics` version `1.3.1`.

Make sure you have them ready either way before the session. You can test it by running the first block in the skeleton Jupyter notebook (with the `import` statements). Note: in Google Colaboratory you can simply install it with `!pip install torchmetrics`. We will in principle not need to use GPUs as we will not be performing any training, so you can choose any hardware you prefer in Google Colaboratory, with runtime `Python 3`.

A characteristic of `PyTorch` is the use of specialised data structures called `Tensors`. They are similar to arrays and matrices (similar to structures in `Numpy`) that can run on GPUs and are optimised for automatic differentiation. They are used to encode the inputs, outputs and parameters of a model. Basic operations we will be doing with `Tensors` include:

- 1) Convert from `Tensors` Numpy arrays.
- 2) Send to GPU or other devices.
- 3) Retrieve scalar values of the tensor (e.g. Python float).

```
# 1 - Convert to Tensor
my_array = np.array(my_data)
my_tensor = torch.from_numpy(my_array)    # or:
image_tensor = torch.Tensor(image_array)
# 2 - Send to device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
tensor = tensor.to(device)
# 3 - Retrieve scalar value
tensor_value = tensor.item()
```

Documentation for `PyTorch` can be found in `https://pytorch.org`. Adapted from PyTorch > Tutorials > Tensors.
Note: For the bonus question you will also need to install `OpenCV` if you are not using Google Colaboratory.

Non-programming exercises

1 - Edge detection

These are the first rows of the matrix used in one of the examples in the Lecture 7:

$$\begin{pmatrix} 1&1&1&1&1&2&2&2&2&2&2&2&1&1&1&1&1 \\ 1&1&1&1&1&2&2&2&2&2&2&2&1&1&1&1&1 \\ 1&1&1&1&1&2&2&2&2&2&2&2&1&1&1&1&1 \\ 1&1&1&1&1&2&2&2&2&2&2&2&1&1&1&1&1 \\ 1&1&1&1&1&2&2&2&2&2&2&2&1&1&1&1&1 \end{pmatrix}$$

Apply to this matrix, using no padding and a stride of one:

(a) The Sobel filter in x, $S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$

(b) A similar filter we can use for edge detection is the Laplacian $\varDelta = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Do the results fit what we have seen in the lecture? How are they helpful to detect edges?

The result is:

(a) With the Sobel filter in x: $\begin{pmatrix} 0&0&0&4&4&0&0&0&0&0&-4&-4&0&0&0 \\ 0&0&0&4&4&0&0&0&0&0&-4&-4&0&0&0 \\ 0&0&0&4&4&0&0&0&0&0&-4&-4&0&0&0 \end{pmatrix}$

(b) With the Laplacian filter: $\begin{pmatrix} 0&0&0&1&-1&0&0&0&0&0&-1&1&0&0&0 \\ 0&0&0&1&-1&0&0&0&0&0&-1&1&0&0&0 \\ 0&0&0&1&-1&0&0&0&0&0&-1&1&0&0&0 \end{pmatrix}$

Both filters identify two vertical edges that do indeed exist in the original image, between the 5th and 6th columns and between the 12th and 13th columns. The Sobel filter shows the edge as columns with a numerical value either positive for the first edge (that moves from pixel intensity of 1 to pixel intensity of 2) or negative for the second edge (that moves from pixel intensity of 2 to pixel intensity of 1). The Laplacian filter shows edges through a change of sign or zero-crossings points.

---

2 - Transposed convolutions

Calculate the padding and zero-insertions (p', z) for the convolution of a 3×3 kernel ($k' = 3$) over a 3×3 input ($m' = n = 3$) using $s' = 1$ that is equivalent to the transpose of convolving a 3×3 kernel ($k = 3$) over a 5×5 input ($m = n' = 5$) padded with $p = 1$ and stride $s = 2$.

The padding and zero-insertions (p', z) should be $p' = 1$ and $z = 1$.

(TURN OVER

3 - Dice Similarity Coefficient and soft Dice loss

Given the following matrix of predictions, A:

$$A = \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.35 & 0.35 & 0.35 & 0.35 & 0.35 & 0.35 \\ 0.7 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.8 & 0.8 & 0.8 & 0.8 & 0.8 & 0.8 \\ 0.9 & 0.9 & 0.9 & 0.9 & 0.9 & 0.9 \end{pmatrix}$$

And the corresponding ground truth labels, B:

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Compute the value of the Dice Similarity Coefficient (DSC, also called Sørensen–Dice coefficient):

$$DSC = \frac{2|A \cap B|}{|A| + |B|}$$

where $|A \cap B|$ should be the sum of the element-wise multiplication of the two matrices A and B, and $|A|$ ($|B|$) should be the sum of elements in matrix A (B).

The value of $|A \cap B|$ is 14.4. On the other hand, $|A| = 17.1$ and $|B| = 18$. Therefore, $DSC = 0.82$ or 82%. A decent score! Can you try to code up this DSC calculation?

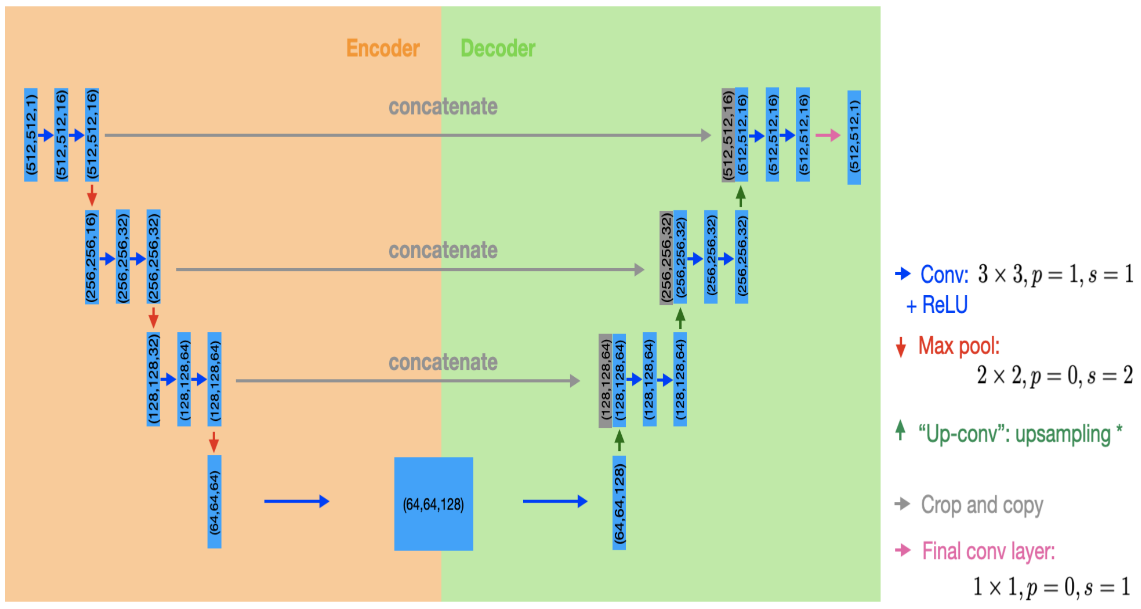

Figure 1: Architecture of a simple UNet for this exercise.

Programming exercises

Simple 2D UNet

For the programming part, you will write the architecture of a simple 2D UNet following the diagram in Fig. 1 and will cross check your implementation by loading a pre-trained model with the same architecture. There are different ways to build this architecture, here we intend you to follow an example of such so that you can load a pre-trained model, so it will be using the suggested `PyTorch` methods. Follow these steps:

1. Write a simple UNet architecture with 3 contracting, 1 middle-expanding, 2 expanding blocks and one final block like in Fig. 1 using `PyTorch`. Use only `nn.Conv2d, nn.ReLU, nn.MaxPool2d` and `nn.ConvTranspose2d.`

   ***Encoder**: Define a convolutional block (* `conv_block` *) that will be repeated in each layer of the encoder. Define as well a maxpooling block (* `maxpool_block` *) that will also be repeated when necessary during the encoding (downsampling path). Hint: use* `torch.nn.Conv2d` *and remember you need to add by hand the activation function (use ReLU).*
   ***Middle**: define the middle or bottleneck part of the architecture.*
   ***Decoder**: Define the transposed convolutional blocks (* `transposed_block` *). Hint: use* `nn.ConvTranspose2d`*.*
   ***Final layer**: Define the final convolutional layer.*
   ***Forward**: Remember you need to define the forward pass of your class, with how each block will be called.*
   You may use Pratical3_empty.ipynb to get started.

(TURN OVER

***Note:** We will use the `PyTorch` implementation of *transposed convolutions* (`nn.ConvTranspose2d`). This is for practical purposes (as the method exists and it's widely used) however notice that the implementation is different to what we have seen in the lecture, and might seem counterintuitive with respect to the general theoretical arithmetics of transposed convolutions that we have discussed. A useful way to understand how this works in addition to the PyTorch documentation (https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html) is through graphic examples (like this one: https://stackoverflow.com/questions/69782823/understanding-the-pytorch-implementation-of-conv2dtranspose). You can compare to the general case we discussed, for example through the illustrations here: https://arxiv.org/pdf/1603.07285.pdf.

2. Add batch normalisation (`torch.nn.nn.BatchNorm2d`) to your UNet network.

3. Add dropout regularisation (`torch.nn.Dropout2d`) of 50% to your UNet network.

4. Test your network is correctly constructed by loading the pre-trained model available in GitHub. This model has been trained with the architecture in Fig 1 and the cases of LCTSC you used in Practical2 (Cases 0-7 for training and 8-11 for testing).

   [*Hint: Define the model using your architecture and load the pre-trained model as* `state_dict` *:* `model = SimpleUNet(in_channels, out_channels)` `model.load_state_dict(torch.load('XXXX.pt', map_location=torch.device(device)))`]

5. How many parameters does your UNet have?

   [*Hint: Use* `from torchsummary import summary`]

6. Visualise some 2D slices of the test dataset in GitLab and their prediction, obtained by evaluating the UNet model on the slice. [*Hint: You may want to use the* `PyTorch` *methods* `TensorDataset, DataLoader`]

7. [BONUS] Apply the processing-based Sobel algorithms to the same 2D slices.

In addition, we will discuss about interpretability during the session, and about how to submit jobs to the HPC.

The solution can be found in the Pratical3_solution.ipynb notebook uploaded to Moodle.

```
Total number of parameters: 536,913
```
