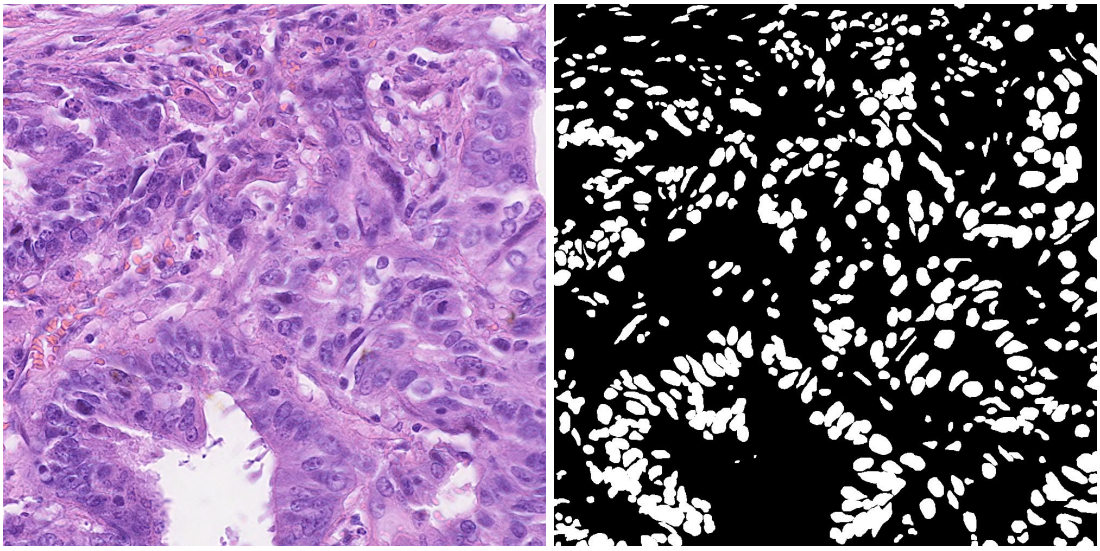
IN4310/3310 - Mandatory 2. Deadline: 29th April 11:59pm

Dhananjay Tomar

Introduction

The aim of this mandatory is to familiarise yourself with creating a neural network, writing its forward pass and writing a training loop to train the model. You will also visualise the outputs of the model. We will work with a nuclei segmentation dataset: CoNSeP. The aim is to generate a segmentation mask for all the nuclei for the provided input image. Nuclei segmentation is a common task in digital pathology. It enables further analysis of the tissue samples.



(a) An H&E stained tissue (b) Segmentation mask for nuclei

Figure 1: An image and its corresponding segmentation mask from the CoNSeP dataset. This image is of size 1000×1000 and has been divided into smaller (overlapping) tiles of size 224×224 for this assignment.

This is an individual exercise, and collaborating with your colleagues is not permitted. Please do not copy from others or distribute your work to others. Your work should be your own.

Dataset

The CoNSeP dataset contains images of size 1000×1000. Since these images are too big to train on, the common practice is to divide the image into overlapping tiles and train on them instead. We have already tiled all the images, and each tile is of size 224×224. The code we wrote for tiling the images can be found in `utils.tile_dataset` file. The data is available on the ML nodes in the following directory:

`/itf-fi-ml/shared/courses/IN3310/mandatory2_data`

or as a zip file (mandatory2_data.zip) in the parent directory (IN3310) for you to download.

We have also uploaded the dataset on Google Drive here. You'd need this link if you'd like to use

Kaggle or Google Colab for Tasks 2 and 3 (check the GPU resources section and watch the videos linked there).

Task 1: Decoder implementation

Your 1st task is to implement the decoder of a modified U-Net in **`resnet_unet.py`**. You should already know U-Net from the Image Segmentation lecture and the corresponding weekly exercise. For this mandatory, we will modify the U-Net by equipping it with two encoders instead of one. Why two encoders for one image? Well, H&E stained slides have two stains: Haematoxylin and Eosin. The hematoxylin stains cell nuclei a purplish blue, and eosin stains the extracellular matrix and cytoplasm pink. There are some algorithms that try to separate these stains in the images. We will use one such method that (hopefully) highlights the nuclei and thus makes it easier to segment it out. But, experimentally, it's better to have both the original and stain-separated images as input to the network. The decoder would then use inputs from both the encoders to output a segmentation mask.

We will use ResNet18 as the encoder. PyTorch provides ResNet's implementation with the option to initialise the weights from a ResNet trained on the ImageNet dataset, so you do not need to implement ResNet18. We provide an Encoder class (a wrapper around ResNet18) whose output is a list containing the outputs of five ResNet blocks.

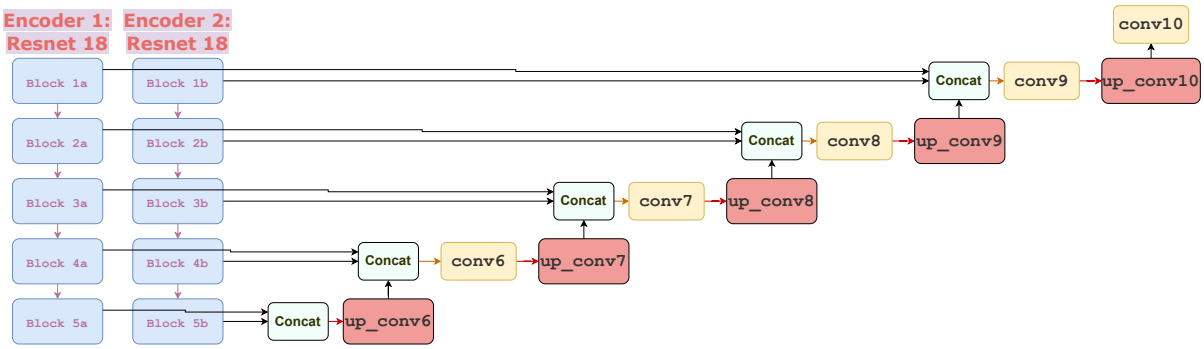


Figure 2: The architecture of the modified U-Net. All the arrows indicate which layers/blocks a given layer receives input from. Three incoming arrows imply that the number of input channels = the sum of output channels of the three incoming arrows. Concat = Concatenate the input using **`torch.cat`**

Part a

The first subtask is to create the layers in the decoder. We've provided two helper functions: **`double_conv`** and **`up_conv`** to create these layers.

Step 1: Find out the number of channels in the output blocks of the encoder. You can pass an image (from the train set) as input to the encoder and print the shape of each item in the output.

Step 2: Initialise the layers in the `__init__` function. Please refer to Figure 2 to calculate the total number of input channels. The number of output channels can be found in the code.

Part b

Implement the forward pass of the decoder based on Figure 2. Note that the assert statement in the code is there to help you check whether your output has the correct shape. Once you know it's correct, you can remove it.

Part c

The second subtask is to initialise the encoders and the decoder in **`class TwoEncodersOneDecoder`** and implement its forward pass. The forward pass should run encoder1 with x as input, encoder2 with h_x as input and finally, the decoder with the encoders' outputs as the inputs.

Task 2: Training loop

Your 2nd task is to fill in the training loop and the evaluation function in **train.py**. We will use two loss functions to train the model. One is the usual binary cross entropy (BCE) loss. The second one is the dice loss. Dice is one of the commonly used metrics for evaluating image segmentation models. It is also used as a loss function in medical image segmentation because training on the metric you're trying to improve directly makes sense.

Part a

Fill out the training loop in the **def** `train` function:

Step 1: Move the tensors to the GPU.

Step 2: The stain-separated image tensor **h_x** containing only the haematoxylin stain has just one channel (instead of three). So, you need to convert it to a tensor with three channels by repeating/copying that one channel, i.e., all three channels should be the same. You can either literally copy the data or use PyTorch's **expand** function that virtually achieves the same thing but without copying data.

Step 3: Run the model to get the predicted segmentation masks for the batch.

Step 4: Compute the BCE and dice losses. Add these two losses to get the total loss.

Step 5: Run the backward pass to compute gradients.

Step 6: Finally, call the optimiser to update the model parameters.

Part b

Fill out the remaining code in the **def** `eval_dice_with_h_x` function. Most of the code here can be copy-pasted from your code for Part a.

Task 3: Run experiments and visualise results

Part a

Now that you've implemented the U-Net and filled in the code for training and evaluating the model, it's time to start training it finally.

If your implementation is correct, your best dice score on the validation subset should be somewhere between 0.83 and 0.85. Feel free to experiment with all the hyperparameters and data augmentation used (both inside the dataset class and outside).

The **train()** function will also generate plots for the dice loss and the BCE loss.

Part b

Call the `eval_dice_with_h_x()` function for the test set and report the dice score your model achieves on the test set. You can run inference on the CPU if you like, but you need to modify the code.

Part c

Finish the TODOs in **visualise_masks.py**.

Call the function with the required parameters to save the visualisations for the test set. Each image will have three columns: 1) the original image, 2) the predicted mask, and 3) the ground truth segmentation mask. You can also run the model locally on the CPU for this task. Modify the code accordingly.

What to submit?

Task 1: For task 1, submit your **resnet_unet.py** file.

Task 2: For task 2, submit your **train.py** file.

Task 3: For task 3, submit your trained model, i.e., the **TwoEncodersOneDecoder_consep.pth** file. Put the following in a PDF file:

1. The loss plots generated after the training loop.
2. Any five good examples of images generated in Task 3 Part c. If you see any bad predictions, put all of them (max. five) as well.
3. Report the dice score achieved on the test set.
4. (Optional) If you tried modifying the hyperparameters and other things, report on what you tried and how that impacted your validation score and/or the visualisations on the validation subset.

How to submit?

Use Devilry to hand in your project work. Use the link devilry.ifi.uio.no, and your UiO username and password to log in. Upload only a .zip compressed file. All your files (scripts, report, files, and model) should be zipped into a file, and please rename it with your username. In the past, some students have faced difficulty uploading the zip file to Devilry because of the size of the model. If so, try uploading your .zip file and the model separately.

Our comments on your project, approval or not, corrections to be made, etc., can be found under your Devilry domain and are only visible to you and the course teachers.

GPU Resources: ML Nodes

You can run your code on your own computer (if you have a GPU) or on the GPU servers. The university has reserved the computing cluster ml9 for this course. Other ml-nodes (e.g. ml6) are also available for use, but not exclusively to us.

We cannot log in directly to the GPU servers, so we first must log in to the login nodes. To log in to the login nodes, use ssh:

```
ssh <user-name>@login.ifi.uio.no
```

To login to the GPU server from the login nodes, use ssh again:

```
ssh <user-name>@ml9.hpc.uio.no
```

Each node has four GPUs, each with 11 GB of GPU RAM. The critical resource here is the GPU RAM. If your code uses more than is available, it will end with a memory allocation error. Use the **watch -n 0.1 nvidia-smi** command to check which GPUs are in use and how much memory is available on those GPUs. After logging in to the GPU server, you must load Python and the libraries needed for the project before you can run any code.

module load is the command for loading any modules.

The module we are using is **PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised**.

To load the module, run the following:

```
module load PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised
```

Do **module list** to see which modules are loaded. And to remove all loaded modules, do **module**

purge. You can check if Python is loaded by running **python --version**, or **which python**. To start running a script, use the following command:

```
CUDA_VISIBLE_DEVICES=<GPU_INDEX> python your_script.py
```

where **<GPU_INDEX>** is the GPU number (0 to 3 if the server has 4 GPUs) you want to use. This command will, however, end when you log out of ssh. To prevent that from happening, you can do the following:

```
CUDA_VISIBLE_DEVICES=x nohup python yourscript.py > out1.log 2> error1.log &
```

or better yet start a new Linux Screen with the **screen -S <screen_name>** command and run your code in there. You can read this tutorial or watch this YouTube video to familiarise yourself with the basics of using screen.

IMPORTANT: Ensure that you only have one main Python process running at a time. Run **pstree -p -U $USER** to check your process tree. When you're training a model, this should show you a tree with only one main Python process and some other processes spun by it as branches of the tree. If you see some old Python process you ran, kill that stale process. To kill a process, do:

```
kill -9 <process-id>
```

GPU Resources: Kaggle and Google Colab

Since the ML nodes have only four to eight GPUs, it's possible that you don't get to use them when you'd like to. The GPUs might also get extra busy as the assignment deadline approaches. Therefore, it might be a good idea to use Kaggle or Google Colab instead.

You will find notebooks for Kaggle and Colab in the code: `kaggle.ipynb` and `colab.ipynb`. To get started, you can upload(import) these notebooks on the respective platforms. But please note that:

1. You still need to submit train.py. So if you use these notebooks, copy-paste the code into train.py for submission.

2. These notebooks are there primarily for Task 2 and 3. So, when uploading the code to the platforms, upload a finished resnet_unet.py file.

3. The notebooks may disconnect in 10-20 minutes if the tab is inactive, so keep checking the tab once you start training on the GPUs there.

4. Prefer Kaggle over Colab as Colab lets you use GPU for a very limited time (a lot less than Kaggle) and doesn't give you any details on how much you have used, how much more you can use and when will you be able to use a GPU again. Kaggle, on the other hand, allows 30 hours of GPU usage per week and shows you your usage.

You will find recorded videos on how to use these platforms on Panopto:

Tutorial on how to use Kaggle (kaggle.com) for this assignment: https://uio.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=7074ec73-e39b-4c2b-970e-b13000fa10ee

Tutorial on how to use Colab (colab.research.google.com) for this assignment: https://uio.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=85ec3c4d-d001-4d40-8666-b130010c7bb7