Mandatory Exercise 1, deadline 15th March 11:59pm

Deep Learning for Image Analysis, IN4310/IN3310

Department of Informatics, University of Oslo, Norway

Feb 28th 2024

The main aim of this exercise is to become familiar with creating data loaders for some image data, training a deep neural network and getting some interpretation of that training, and understanding how to evaluate the model on an unseen set. We will be working on a classification problem of nature/city scenes. There are six different classes in this dataset, with a total of 17034 images. See the figure below for some image examples.

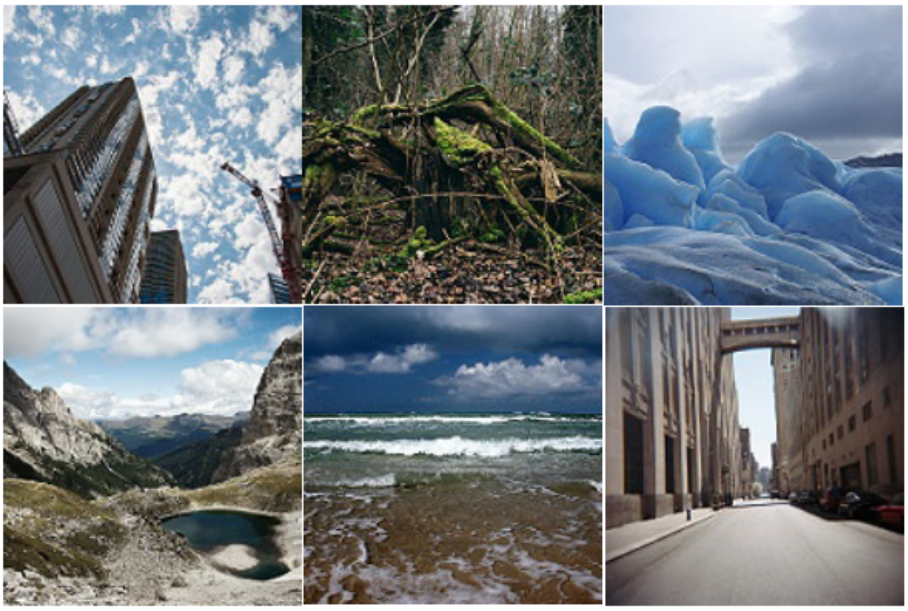

Figure 1: Examples of the six classes: buildings, forest, glacier, mountain, sea, street. Buildings and street classes can have some ambiguity.

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.

1 Task 1:

Part a): The first part is to split the data into three sets: training, validation, and test. Split each class separately so that there is an equal percentage of each class in all three sets. The number of total images in the validation is to be roughly 2000, in the test set 3000, and the remainder for the training set. It does not need to be exact. This is called a stratified split. The scikit-learn library, for example, has functions for this.

The images are in `/itf-fi-ml/shared/courses/IN3310/mandatory1_data`, and the subdirectories define the labels. For copying the data off the ml-nodes, the data is stored in `/itf-fi-ml/shared/courses/IN3310/mandatory1_data.zip`.

Write code to verify that your sets are disjointed. Disjointed, in this case, means that no file is found within more than one set.

Hints: os.path has useful functions for concatenating and splitting filesystem paths. If you save filenames with paths for your three sets, you should save the path relative to a dataset root path onwards. This is to ensure that if the dataset moves, your files/filenames are still usable with the new dataset root location.

Part b): Write code for the training, validation, and test dataloaders. Please make one root path for the dataset, this makes it easier for us to check/debug your work. If there are multiple paths to the dataset we need to change, it becomes tricky to change them all.

Part c): Choose a model for training. A ResNet type model is fine, but the choice is yours. We will be finetuning the model, not training from scratch. Also choose an appropriate loss function for the multi-class classification task.

Part d): Write code to calculate the accuracy per class and the average precision (AP) per class. It's not necessary to report this for the training, just validation and testing. Also report the accuracy and average precision averaged over all the classes, for the latter this is called the mean average precision (mAP). When you compute the AP for a class you will need to modify the label so that it is 1, if the image is labelled with the class of interest, and 0 otherwise (from another class).

Part e): Training your model. Your Python code should be able to be run without extra parameters on the command line, just python my_code.py. We recommend setting the seeds (pytorch, numpy) to make your work reproducible. Please adjust your batch size so that your code takes no more than 2 GB of GPU memory. You can use the following commands to check the memory your code uses:

```
ps -axu | grep <user-name> | grep python
```

To get the job process id use:

```
nvidia-smi
```

Train your model with three different hyper-parameter settings. The hyper-parameter could be the learning rate, optimizer, data augmentation parameters, or others. Save the model with the best performance on the validation set. The performance here means the mAP. Record and plot the mAP and mean accuracy

per class for each epoch. Record the training and validation loss and plot the losses with the number of epochs on the x-axis. Try to interpret what is going on in the model from the loss curves and metric scores.

Part f): Predict on the test set, compute the mAP and mean accuracy per class, and save the softmax scores to file. For three classes of your choice, show ten images of the worst and ten of the best images according to the softmax score. Write code to load the test set, predict on the test set, and then compare these against your saved softmax scores. There can be some tolerance between the two. Please use relative paths from the main Python files for loading the scores, model, etc. Only use an absolute path for the dataset root.

2 Task 2: Feature Map Statistics

This next task uses the same dataloader that you created in task 1. We will be using forward hooks to get the feature maps from some locations in the model network. See this page for more information on hooks `https://pytorch.org/docs/stable/generated/torch.nn.modules.module.register_module_forward_hook.html`.

Part a): Choose five feature maps as the outputs of the modules. The following loop allows you to iterate through all the modules in the network:

```
for name, mod in model.named_modules():
```

Part b): For these five feature maps, compute the percentage of non-positive values (zero is non-positive) over all the spatial dimensions and channels. Report the average percentage over 200 images. In networks that contain the ReLU activation function, the outputs are usually zero. With other functions, ELU, GELU, and SELU, the output of the activation can be negative.

There are two ways to do this. The first is that you can store the whole feature map and then compute the statistic afterward. When saving feature maps, the hook will need information about the list of image filenames or batch index to be used to save information that depends on a mini-batch. If you want to provide the hook with a batch index, then you can write it into the pytorch module.

```
for batch_ind, data in enumerate(dataloader):
    images = data['img']
    labels = data['labels']
    for nam, mod in model.named_modules():
        if nam in selectedmodules:
            mod.batchindex = batch_ind
```

Then, within the hook, you can reference the batch index. Alternatively, instead of saving the feature maps, you can compute the statistic inside the hook and save the statistic. You could also write the current statistic back into the pytorch module. For this, you need the current number of data used so far. Knowing the number of data used, you can iteratively update the average:

$$m_{t+1} = \frac{m_t * n_t + u_{step} * n_{step}}{n_t + n_{step}}$$

$$n_{t+1} = n_t + n_{step}$$

If you choose the former, please delete your feature maps at the end of the project, as this will use a lot of storage space on the servers.

3 Task 3 (For master students only):

In this task, we want to visualize the separation of the samples before the last linear layer - again by using a hook.

Part a): We will use PCA to reduce the high-dimensional features down to two so that we can plot it in a 2-D scatter plot. We want to compare the separation of samples in the validation set using your best model and using a model not trained on this dataset. The PCA transform needs to be done with all samples, and not in mini-batches. Hence, you can either store the features every batch, or write them to the module.

Part b): Make a scatter plot with points color-coded by their label (include a legend). Do this for both models, your best model and an untrained model. Try to interpret what is shown in these plots, what your best model is doing, and what the untrained model is not.

4 GPU Resources and Python interpreters

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 and 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 of the nodes has eight 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. With a ResNet-18 model, your code will consume less than 2 GB with a batch size of 16. As mentioned earlier, use nvidia-smi command to check which GPUs are in use, and how much memory is available on those GPUs. If there are 2 GBs available, you can use it. After logging in to the GPU server, before you can run any code, you will need to load Python and the libraries needed for the project.

`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, do:

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

To see what modules are loaded, do `module list`. And to remove any loaded modules, do `module purge`. You can check if Python is loaded by doing `python --version`, or `which python`. To start running a script, use the following command:

```
CUDA_VISIBLE_DEVICES=x python your_script.py
```

`x` is the GPU that 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 &
```

Let's go through this command: `nohup` starts the command without hangup, `> out.log` redirects the output of the script to the file `out.log`, and `2> error.log` redirects the error messages in the same way, and the & symbol places the job running in the background.

To kill a process, first show the processes and IDs that you are running. Do this with the following command:

```
ps -u <user-name>
```

And doing:

```
ps -u <user-name> | grep -i python
```

will only show the Python processes you are running. Both of these commands will show the process id. To kill a process, do:

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

5 Running and debugging with a remote interpreter

Debugging your code with the method above can be time-consuming as you may need to change the code and deploy/upload it to the server manually with scp/rsync each time. You can use a remote interpreter, i.e., debug your code in an IDE (integrated development environment) on your computer, but the code is actually running on the GPU server. We recommend PyCharm. You can use others, but the group teachers may not be able to give you support for the debugging. To do this, we will create a bash script that will load Python and the modules needed (from the lmod system). We will then point to this bash script (as our Python interpreter) in PyCharm or another IDE of your choice. Your code can then be deployed and debugged without having to use the method above.

To set this up follow these steps:

5.1 Step-by-step to setup the remote interpreter

- Open one terminal and run the following command:

  ```
  ssh -J username@login.uio.no username@ml9.hpc.uio.no
  ```

  This will take you directly to the ML9 node.

- While in ML9, create a new bash file, e.g. using `vim`, and paste this content inside:

  ```
  #!/bin/bash
  source ~/.bash_profile
  module load PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised
  # Run the Python interpreter with the passed arguments
  exec python "$@"
  ```

  Check this link on how to use vim to create and save files.

- Make that script executable by you, by running `chmod 700 your_bash_script.sh`

- Open another terminal and run the following command:

  `ssh -L 6000:ml9.hpc.uio.no:22 <user-name>@login.ifi.uio.no`

  This will allow you to tunnel ssh from your local machine to ML9 through the login node, so that you can use the remote interpreter.

- On the bottom left corner of PyCharm, click on the interpreter you have, e.g. "Python 3.x", "Add New Interpreter", "On SSH..."

- For host, type "localhost", for the port, use 6000 (the same number we used in the ssh command above), and enter your username, then click Next.

- PyCharm will perform some checks, when it finishes click "Next"

- For the Environment, choose "Existing", then for the interpreter, click on the three horizontal dots on the right, and brows through ML9 and choose the bash script that you've created in step 1. The path will be something like:

  `/itf-fi-ml/home/username/your_bash_script.sh`

- For the sync folders, click on the browse icon on the right, go to your username directory on ML9, create a folder there, e.g. `mand1`, and then choose that folder. Then, Local Path should be your root directory where your local project is, and Remote Path should be the folder you created. This will allow PyCharm to sync files between these two directories. Then, click Create.

With these steps, you should be able to upload any python file you have on you local machine, by right clicking anywhere inside the file, Deployment, "upload to ...". Afterwards, you should be able to run files remotely from PyCharm for debugging purposes.

6 Submission of your work

This is an individual exercise, and collaborating with your colleagues is not permitted. Please do not copy from others or distribute code to others. **Your work should be your own**. Suspicion of cheating may also arise if your work is generated by AI tools such as **Chat GPT or similar**.

Place all metrics we have asked you to report, plots, and comments/interpretations in PDF file. Use Devilry to hand in your project work. Use the link `devilry.ifi.uio.no`, and your UiO username and password to login. 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. It has happened in the past that some students had difficulty uploading the zip file to devilry because of the size of the model. If this is the case, try uploading your .zip file and the model separately.

Comments from us 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 teachers of the course.