Mandatory Exercise 1, deadline 14th March 11:59pm

Deep Learning for Image Analysis, IN4310/IN3310

Department of Informatics, University of Oslo, Norway

Feb 28th 2025

The primary goal of this exercise is to gain hands-on experience in creating data loaders for image data, training a deep neural network, specifically a ResNet neural network, and interpreting the training process. In addition, our goal is to understand how to evaluate the model on an unseen data 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.

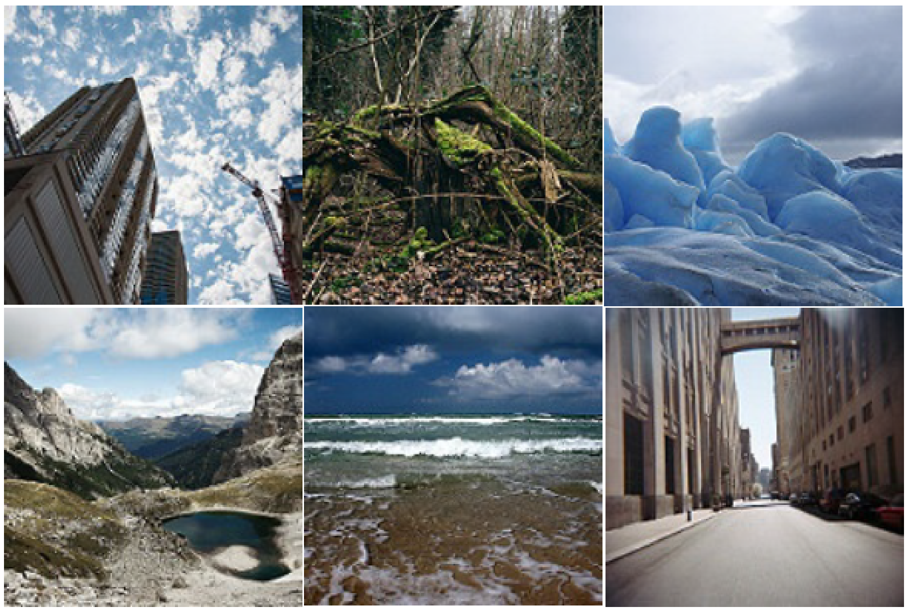


Figure 1: Examples of the six classes: buildings, forest, glacier, mountain, sea, street. Buildings and street classes can have some ambiguity.

This is an individual exercise, and collaboration 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: Dataset loading

Part a): The first task is to split the data into three sets: training, validation, and test. Ensure that each class is split separately so that there is an equal percentage of each class in all three sets. This is called a stratified split.

- **Data Distribution:**
  - Validation set: Approximately 2,000 images.
  - Test set: Approximately 3,000 images.
  - Training set: The remainder of the images.

  Note: The exact numbers do not need to be precise, but aim to keep the proportions consistent.

You can use libraries like `scikit-learn`, which have built-in functions to perform a stratified split.

- **Data Location:**
  - The images are located at `/itf-fi-ml/shared/courses/IN3310/mandatory1\_data`.
  - Subdirectories within this directory correspond to the different class labels.

  To copy the data off the ml-nodes, use the data stored at `/itf-fi-ml/shared/courses/IN3310/mandatory1\_data.zip`.

Part b): Create a solution to verify that the dataset splits are disjoint. Ensure that no file appears in more than one of your training, validation, or test sets.

Hint: `os.path` has useful functions for concatenating and splitting file system 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 c): Develop and implement dataloaders for training, validation, and test sets. 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 that we need to change, it becomes tricky to change them all.

2 Building and training ResNet architectures

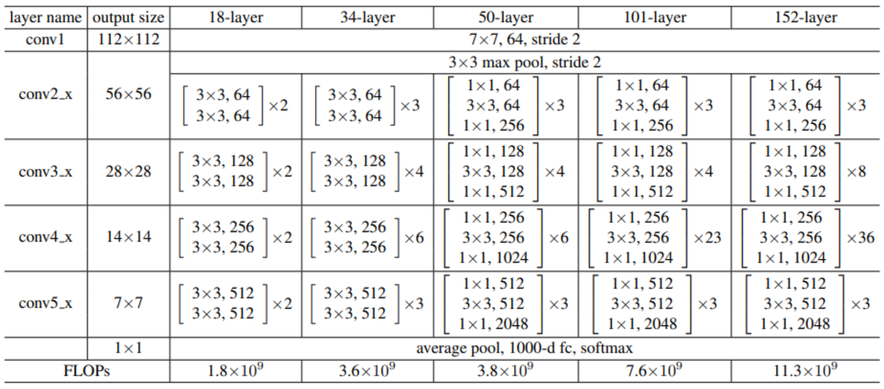
2.1 Implementing ResNets

This section is dedicated to exploring and experimenting with various Residual Neural Network (ResNet) architectures, specifically focusing on ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152. While detailed resources, including original papers, are available for these architectures, this assignment will emphasize a solid understanding of the core concepts and primary building blocks rather than an exhaustive study of each model.

ResNets are a type of Convolutional Neural Network (CNN) known by its organization into multiple "blocks." These blocks are categorized into two types: basic blocks and bottleneck blocks. Basic blocks are simpler and typically used in shallower ResNets like ResNet-18 and ResNet-34. The bottleneck block, on the other hand, is more complex and is used in deeper architectures like ResNet-50, ResNet-101 and ResNet-152.

What sets ResNets apart from other CNNs is the use of residual or skip connections. These connections address the challenges of training deep neural networks by creating shortcuts that bypass one or more layers, linking the input of one block directly to the output of another. This approach effectively mitigates issues like vanishing gradients, allowing for more efficient training of deeper networks.

Throughout this assignment, you will investigate and compare these ResNet architectures, gaining insights into the different block structures.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2: Architectures for ImageNet. Building blocks with the numbers of blocks stacked

Part a): In this task, you will extend an existing implementation of the ResNet-18 architecture to support ResNet-34, ResNet-50, ResNet-101, and ResNet-152. This exercise will help you deepen your understanding of how the ResNet family of architectures is structured under the hood.

You are provided with some pre-code that is split into two main files:

- ResNet.py: Contains the core structure of the ResNet models.

- ResNetBlocks.py: Contains the completed implementations of two block types:
  - BasicBlock: Used for smaller architectures like ResNet-18 and ResNet-34.
  - BottleneckBlock: Used for larger architectures like ResNet-50 and above.

Familiarize yourself with the already working 18-layer architecture provided in the ResNet.py file. After developing an understanding for the structure of the 18 layer architecture, extend the code to support architectures with more layers.

Using the pre-existing ResNet constructor as a reference, expand the ResNet module to support:

- ResNet-34 (which BasicBlock)
- ResNet-50, ResNet-101, ResNet-152 (using BottleneckBlock).

Figure 2 shows the architecture and provides a high-level overview of ResNet architectures with different numbers of stages.

Part b): Now that you have completed the implementation of the different ResNets, choose one of them to move forward with (or continue with the default, which is the ResNet-18). Use the dataloaders you created in Part 1 to feed the training data into the model of your choosing. Write code to perform the training process, ensuring that the model is optimized over the training data. Make sure to use the validation dataset to monitor performance during training. During training, monitor the model's performance using accuracy on the validation set. This will give you an initial indication of how well your model is learning. Note that in the next section, you will evaluate the model using metrics beyond accuracy.

Important: Do not import any pre-trained ResNet models from PyTorch or any other libraries. Use only the ResNet module from the precode and initialize it according to your needs.

Hints:

- To keep your code organized and maintain separation of concerns, it's recommended that you import the ResNet module into a separate file where you'll write the training code. This approach helps to ensure cleaner, more maintainable code. Ex.

```
from ResNet import ResNet

#initializing
model = ResNet(...)
```

- When training the model remember:
  - to put the model into the correct mode train()/eval()
  - zero out the gradients
  - update optimizer

2.2 Metrics and Performance Evaluation

Part c): To gain deeper insights into your model's performance, you can analyze its effectiveness by calculating the accuracy and average precision(AP) for each class. It's not necessary to report this for the training, just validation and testing. You will use these metrics to evaluate your models performance. This will help you get an overview of how well the model distinguishes between different classes.

In addition, you should compute the accuracy and average precision averaged over all classes; for the latter, this is called 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 d): 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.

Note: When saving your model, keep in mind that saving the entire model can result in a large file size. To be more storage-efficient, it's recommended to save only the model's weights. Later, you can load these weights into your model without having to retrain it from scratch, saving both time and computational resources. Here is a short guide: follow this link

Optional: Feel free to experiment with all the ResNet models and compare results.

Part e): Predict on the test set, compute the mAP and mean accuracy per class, and save the softmax scores to file. 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.

Include in your report

- Loss graph for train and validation set(the axis should be loss vs time(epochs))
- Loss graph for testset (the axis should be loss vs time(epochs))

- Accuracy per class for Validation set
- Accuracy per class for Test set
- mAP for validation
- mAP for Test
- part f), discussing transfer learning

Part f): So far, you have trained a ResNet from scratch, which means that the model has learned only from the specific data set you provided.

In this section, you will be working with a pre-trained ResNet model of your choosing provided by the torchvision library. For the sake of consistency, use a model with the same number of layers as in task b), if you choose to work with a ResNet-18 then import the ResNet-18 model from torchvision. Using this model is straightforward: you need to import it from torchvision, configure it appropriately, and then train it in a manner similar to the model you previously built from scratch. Import it like this:

```
import torchvision.models as models

#initializing a resnet18 model with pretrained weights
model = models.resnet18(pretrained=True)
```

The advantage of using a pretrained model like the ones provided by torchvision is that you can reuse the learned features from a model trained on a large dataset, such as ImageNet, which contains millions of labeled images. This allows your model to start with a strong baseline, instead of learning only on the dataset you provide.

You will need to reshape the last classifier layer to take into account the number of classes in your dataset. The pretrained models provided by torchvision, such as ResNet, are typically trained on the ImageNet dataset, which has 1,000 classes. However, if your dataset has a different number of classes, you must manually adjust the final layer

Train the model on the training set and evaluate its performance on the validation dataset. Next, run the test dataset through the trained model and analyze the results. Be sure to calculate the same metrics as in the previous tasks (AP, mAP) and plot the loss curves for the training, validation, and test sets. The results, particularly the mAP, should show improvement.

3 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`.

Your task is to try to analyze and visualize the feature maps from a convolutional neural network (CNN) during the forward pass of an image. For simplicity, you can use the pre-trained module from torchvision. This exercise will help you understand the internal workings of CNNs, particularly how different layers process input images.

Part a): Identify a few key layers in the pre-trained model that you want to analyze. Consider selecting layers from different stages of the network (early, middle, and late) to observe how the features evolve. As a starting point, you might choose a layer close to the input, one in the middle of the network, and one near the output.

Hint: You if you want to grab the feature maps from one of the ResNet models provided from Resnet.py then you can use the following:

```
layer_names = ['stage1', 'stage2', 'stage3', 'stage4']
```

If you want to see the feature maps from the models provided by torchvision then you can use set this for the layer names:

```
layer_names = ['layer1', 'layer2', 'layer3', 'layer4']
```

Part b): Use PyTorch's forward hooks to capture the output of the selected layers. A forward hook is a function that gets called every time a forward pass is made through the layer, allowing you to capture and store the output (i.e., the feature map) of that layer. When you run a forward pass with an image, the hook will automatically capture feature maps from layers where it is registered.

After registering your hooks, run a few forward passes with different images from your dataset. This will trigger the hooks and store the feature maps for each registered layer. Ensure that your hooks save the captured feature maps in a way that you can easily access them later (e.g., storing them in a dictionary with layer names as keys).

Part c): Visualize the captured feature maps (from the previous task) to see the patterns learned by each layer. You may start by retrieving a batch of images from your dataloader. You will focus on the first 10 images in this batch. For each of these images, pass it through the model to perform a forward pass. During this forward pass, the hooks you registered earlier will capture the feature maps from the selected layers.

After running the forward pass on each image, your task is to visualize and save the feature maps as PNG files. Each image should result in a corresponding set of PNG files that represent the feature maps for the layers you are analyzing. Name each file to reflect the index of the image it corresponds to. By the end of this task, you should have a set of PNG files that show the feature maps for the selected layers across the first 10 images in your batch. This collection of visualizations will allow you to explore the different patterns learned by each layer in the network.

Part d): Comment on your observations, highlighting any differences you notice across the layers. Provide a brief explanation for these differences and consider why they might occur.

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

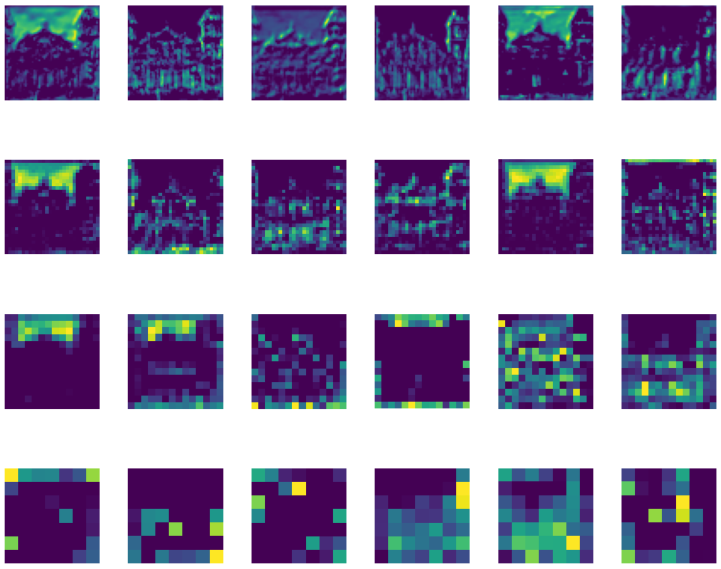


Figure 3: An example of feature maps generated from a single image across four different layers. Each row corresponds to a specific layer, displaying 7 feature maps per layer.

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

Part f): 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.

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. You can use UiO's computing cluster, e.g., ml9 or other ml-nodes.

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.