IN3310 Mandatory 1

March 12, 2025

Håkon Ganes Kornstad

General note for Tasks 1 & 2: The code for these tasks can be found in `ResNetData.py`, `ResNetTrain.py`, `FeatureAnalysis.py` and `plotting.py`. A run of `haakongk_main.py` will generate all the data, plots and csv described in this report, provided that the `mandatory1_data` folder is present together with these runfiles.

Task 1: Dataset loading **a)** After downloading the data from the server, a `ResNetData` class is implemented to organize the data and make a stratified split. After some testing, I choose **65% training data**, **16% validation data** and **19% test data**. The *training set* needs to be big enough so that the model can learn from the data, and generalize on unseen data. The *validation set* is used to keep an eye with the training along the epochs, and also possibly adjust any hyperparameter along the way. It should be smaller, however a too small validation set makes the evaluation unstable. Finally the *test set* should be about the same size as the validation, but I choose it to be a bit bigger.

Instead of creating a folder structure `train, val, test` and copying the files here, `ResNetData` creates the file `annotations.csv`, containing the file paths of each image, along with the class name and stratified split info:

[1]:
```python
import pandas as pd

df = pd.read_csv('annotations.csv')
print(df.head())
```

```
  split                                        image_path  label
0  train  /mnt/e/ml_projects/IN3310/2025/IN3310/Mandator…      0
1  train  /mnt/e/ml_projects/IN3310/2025/IN3310/Mandator…      0
2  train  /mnt/e/ml_projects/IN3310/2025/IN3310/Mandator…      2
3  train  /mnt/e/ml_projects/IN3310/2025/IN3310/Mandator…      1
4  train  /mnt/e/ml_projects/IN3310/2025/IN3310/Mandator…      5
```

b) We now want to assert that there is no leakage of similar images into the different splits. I implement this as a helper function in the `ResNetData` class, namely `_verify_disjoint_splits()`. The function simply reads `annotations.csv` and makes three **sets** of file names based on the respective split. Then `intersection()` is used between them: it should return 0 for disjoint sets. I choose to include this function in the constructor for `ResNetData`, so it will run automatically when creating the data.

c) A standard PyTorch `DataLoader` is then implemented. The constructor again reads annotations.csv, and creates an array of the images we want to include, based on a given split. It also takes in the transforms, and determines whether an *augmented transform* is present. Then, in `__getitem__()`, we split out the `label` information, and `Image.open()` is used on a per-image basis. The transformed image is returned along with the label.

Task 2: Implementing ResNets

a) Please refer to the file `ResNet.py`, where the alterations were done according to the task.

b-c) We can now implement a test training on the default ResNet-1

[2]:
```
from pathlib import Path
from ResNet import ResNet
from ResNetData import ResNetDataPreprocessor, ResNetDataset
from torchvision import transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
from ResNetTrain import train_model

BASE_PATH = Path.cwd()
DATASET_PATH = BASE_PATH / 'mandatory1_data'

# first we instigate a Preprocessor
preprocessor = ResNetDataPreprocessor(base_path=BASE_PATH,␣
 ↪dataset_path=DATASET_PATH)

# we set up the transforms
transform = transforms.Compose([
    transforms.Resize((150, 150)),  # adjusting size
    transforms.ToTensor(),  # converting to PyTorch tensor
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # normalizing for␣
 ↪RGB-bilder
])

# getting the datasets and -loaders
train_dataset = ResNetDataset(preprocessor.annotations_file, BASE_PATH,␣
 ↪split='train', transform=transform)
val_dataset = ResNetDataset(preprocessor.annotations_file, BASE_PATH,␣
 ↪split='val', transform=transform)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

class_names = preprocessor.get_class_names()

# inititating a model
model = ResNet(img_channels=3, num_layers=18, num_classes=len(class_names))
```

```
# setting up the loss function, optimizer and
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# we can now train the model
file_path1 = BASE_PATH / 'resnet1.pth'
train_accs1, val_acc1, map_scores1, class_accs1, train_losses1, val_losses1 =␣
 ↪train_model(model, train_loader, val_loader, criterion, optimizer,␣
 ↪file_path1, num_epochs=3)
```

```
Success! Train, validation and test sets are disjoint
Epoch 1/3, Train Loss: 1.0018, Val Loss: 0.8644, Train Acc: 0.6119, Val Acc:
0.6793, mAP: 0.7492 - Model Saved
Epoch 2/3, Train Loss: 0.7151, Val Loss: 0.7042, Train Acc: 0.7367, Val Acc:
0.7360, mAP: 0.8387 - Model Saved
Epoch 3/3, Train Loss: 0.5989, Val Loss: 0.6195, Train Acc: 0.7806, Val Acc:
0.7740, mAP: 0.8955 - Model Saved
```

d) After this "sneak peek", we are ready to do the full training. After some initial runs to test performance, the following three models are chosen for this task (`batch_size=32` used for all): 1) **ResNet34** with CrossEntropyLoss, Adam optimizer, Learning Rate: 0.001, using basic transforms, batch size 32 2) **ResNet34** with CrossEntropyLoss, Adam optimizer, Learning Rate: 0.001, using a set of augmented transforms (see below), batch size 32 3) **ResNet34** with CrossEntropyLoss, Stochastic Gradient Descent, Learning Rate: 0.005, using the basic transforms, batch size 32

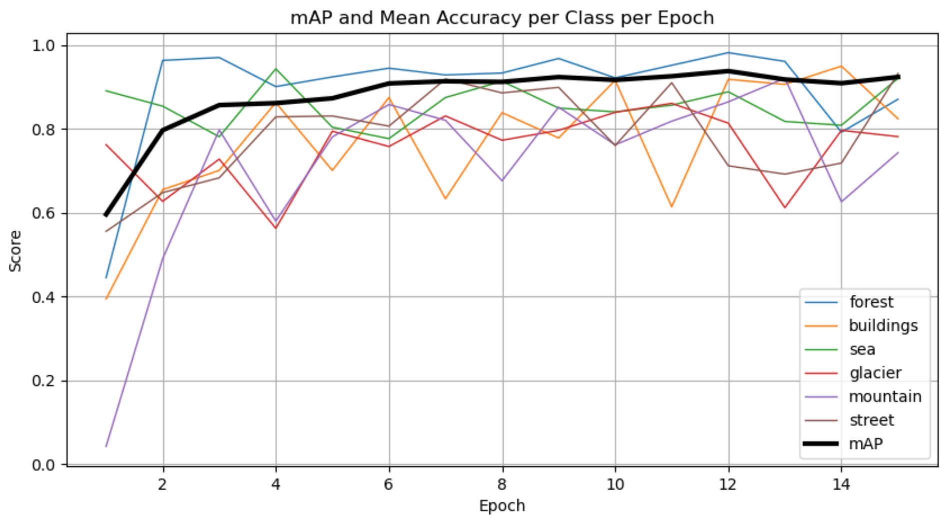
For the augmentation, the intuition was limited on what to use, but in order to experiment a bit, the following `transforms` was set up:

[3]:
```
# data augmentation
augm_transform = transforms.Compose([
    transforms.RandomResizedCrop(150, scale=(0.8, 1.0)), # scaling and cropping␣
 ↪the image
    transforms.RandomHorizontalFlip(p=0.5), # performing a horizontal flip of␣
 ↪50% of the images
    transforms.RandomRotation(15), # random rotation within 15 degrees
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.
 ↪1), # some color augmentation
    # the basics:
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

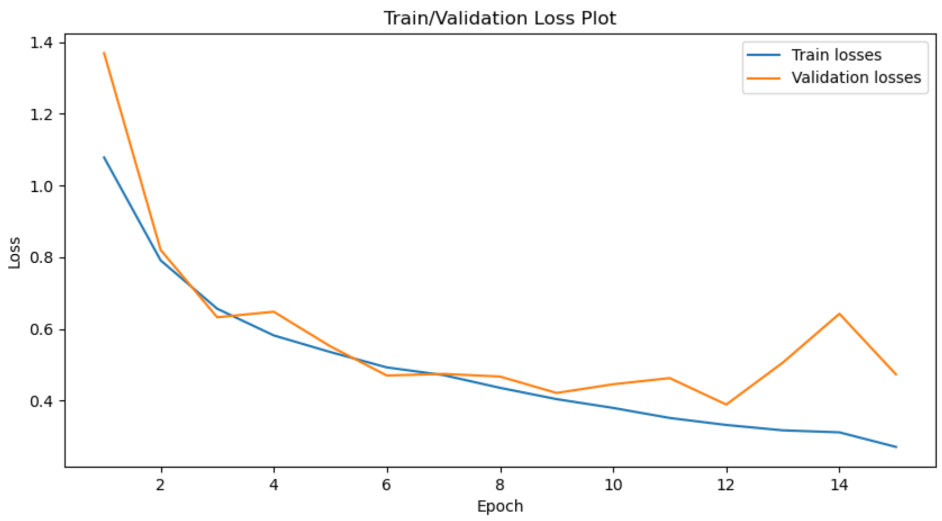
Throughout the training, the command `watch -n 1 nvidia-smi` kept track of the GPU, showing less than 2GB memory consumption at all times for these models.

A "brute force in main" functionality was then implemented to select the best of these three models after running them all, based on the mAP-values. Also, `early stopping` was introduced with 3 epochs set as the threshold: the training would then stop after three epochs if the mAP value had not been increased. Quite consistently, the best performing model turned out to be **model 1** with `maP = 0.9402`.

Mean Accuracy values per Class was recorded during training per Epoch, and a plot shows the variance in the results during training. Initially the classes are trained from a low score, and then are quite unstable for the first 2-6 epochs. At epoch 9-10, the accuracy is perhaps most stable in all of the classes. From a visual inspection, this would be a good place to stop training. However, the model manages to find a better mAP score at epoch 14, before stopping early at epoch 17 after three uneventful runs.



The Train/Validation Loss Plot is classic in its shape, with a validation loss starting out slightly over the training loss. For this particular training the validation loss decreases quite quickly to join the training loss values. Then, around epoch 9-10, the validation loss starts to increase, which is a sign that the model has begun overtraining. Interestingly, the overtraining seems to be evident from where we concluded that the model was most stable in the previous plot: perhaps we should have stopped training at epoch 10.



e) We now performed an evaluation on the test set, by loading the best model with `model.load_state_dict()` and then `evaluate_model()` from `ResNetTrain.py`. This rendered the following output:

```
Test Accuracy: 0.8677
Test Loss: 0.3836
Test mAP: 0.9402
Test Mean Accuracy per Class: 0.8700
```

The assignment then asked for a plot of mAP scores on a per epoch basis for the test run. However, following a discussion on Mattermost, this was deemed unneccessary. This is in line with the practice that the test set should generally be kept in a vault until it is time to do a final test on a promising model. Testing along with each epoch might be good for illustration, however it is “dangerous”, as we risk doing the mistake that the model trains on the test data.

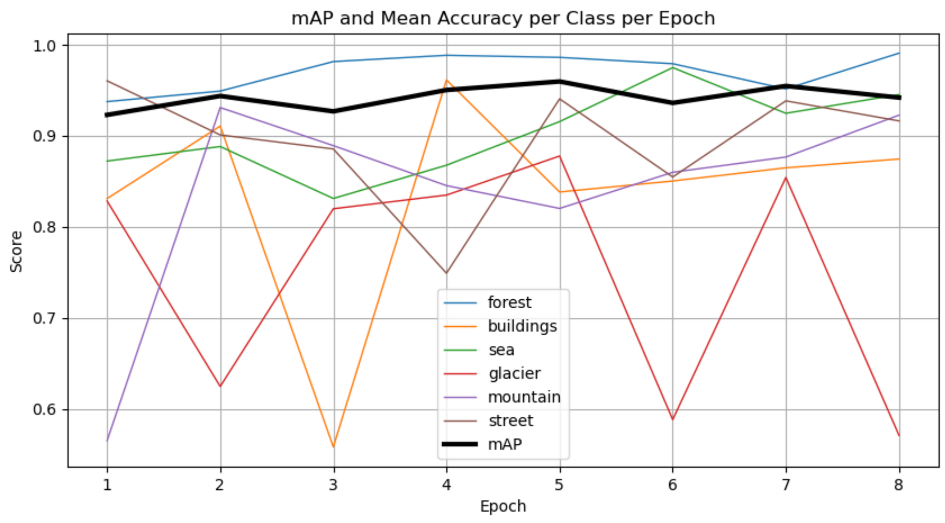
f) A pre-trained PyTorch ResNet was now imported, using the `DEFAULT` weights:

[4]:
```python
from torchvision.models import resnet34, ResNet34_Weights

model = resnet34(weights = ResNet34_Weights.DEFAULT)
```

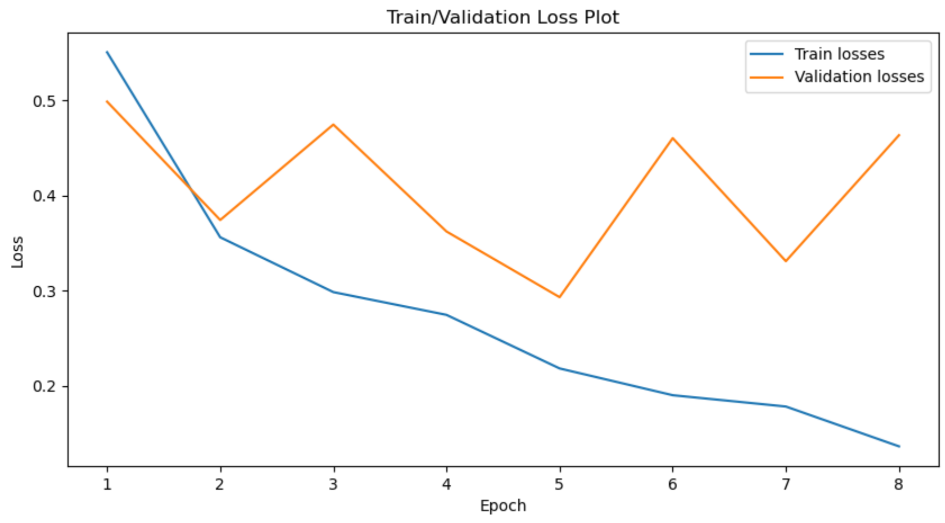
In order to use it for our purpose, the output layer needs to be set to the same size as our classes:

[5]:
```python
num_classes = len(class_names)
model.fc = nn.Linear(model.fc.in_features, num_classes)
```

It was now trained with the same optimizer as our other model, namely `Adam` with `lr = 0.001`, as well as `CrossEntropyLoss`, and the results were plotted like before:



First of all, this pretrained model had a consistently higher mAP score almost from the start, and reached > 0.95 after just 2 epochs. The individual classes seem to have the most stable and overall high result around epoch 5-6.



The Train/Validation Loss Plot confirmed this impression from the visual inspection: from epoch 2, the validation loss starts to creep upwards, indicating that the model is starting to overfit to the noise in the training data.

Evaluation on the test set rendered the following results:

```
Test Accuracy: 0.8689
Test Loss: 0.4526
Test mAP: 0.9408
Test Mean Accuracy per Class: 0.8710
```

This is only slightly better than our “own” model.

Task 2

[6]:
```
from pathlib import Path

from ResNetData import ResNetDataPreprocessor, ResNetDataset
from ResNetTrain import *
from plotting import plot_losses

import torch

from torch.utils.data import DataLoader
from torchvision import transforms

from torchvision.models import resnet34, ResNet34_Weights

import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from PIL import Image

import matplotlib.pyplot as plt
import os

import random
```

a) Identifying which layers in the pre-trained model that was relevant to look at: usually earlier layers in combination with a middle and deep layer. Let’s look at the names first…

[7]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.
  ↪device('cpu')

BASE_PATH = Path.cwd()
DATASET_PATH = BASE_PATH / 'mandatory1_data'

model = resnet34(weights=ResNet34_Weights.DEFAULT).to(device)
model.eval()

# Finding out which layers we have here
for name, module in model.named_children():
    print(name)

# choosing layers to look at
layer_names = ['layer1', 'layer3', 'layer4']
```

```
conv1
bn1
relu
maxpool
```

```
layer1
layer2
layer3
layer4
avgpool
fc
```

b) We should now use PyTorch's forward hooks to capture the output of the selected layers. A `hook_function()` needs to be set up for this purpose.

[8]:
```
feature_maps = {}

def hook_function(module, input, output, layer_name):
    feature_maps[layer_name] = output.detach().cpu()

hooks = []
for name in layer_names:
    layer = dict(model.named_children())[name]  # finding the right layer in␣
 ↪the model
    hook = layer.register_forward_hook(lambda module, input, output, name=name:␣
 ↪hook_function(module, input, output, name))
    hooks.append(hook)
```

After registering the hooks, we were supposed to run a few forward passes with different images from the dataset. To ensure that the hooks are triggered, the feature maps were saved in a dictionary with layer names as keys.

[9]:
```
feature_maps = {}

# fetching data
preprocessor = ResNetDataPreprocessor(dataset_path=DATASET_PATH,␣
 ↪base_path=BASE_PATH)

# defining our standard transform for a dataset
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # ResNet needs 224x224
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# getting tha dataset from our existing code
test_dataset = ResNetDataset(preprocessor.annotations_file, BASE_PATH,␣
 ↪split="test", transform=transform)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=True)  # using␣
 ↪batch_size=1 for single images

# getting feature maps from some images
num_images = 5  # choosing the amount of images to analyse
```

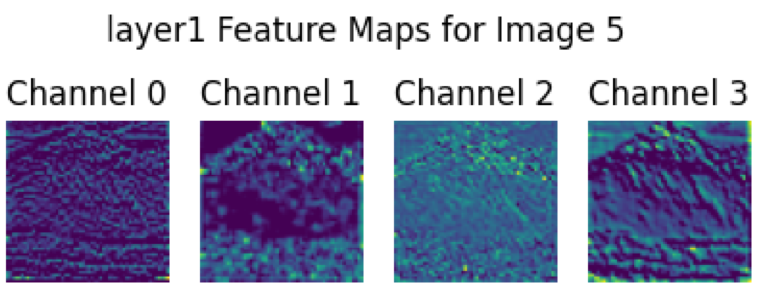
```
model.eval()

with torch.no_grad():
    for i, (image, label) in enumerate(test_loader):
        image = image.to(device)  # if we have GPU, move the image there
        model(image)  # a forward pass will activate the

        print(f"Feature maps stored for image {i+1}")

        if i+1 >= num_images:
            break
```

```
Success! Train, validation and test sets are disjoint
Feature maps stored for image 1
Feature maps stored for image 2
Feature maps stored for image 3
Feature maps stored for image 4
Feature maps stored for image 5
```

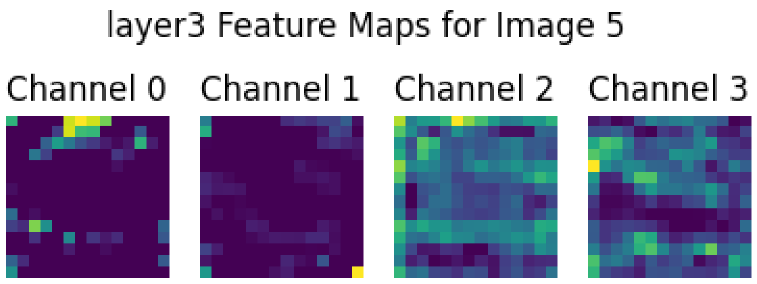
c) Visualising the layers in the feature maps

[10]:
```
for i, (layer_name, feature_map) in enumerate(feature_maps.items()):

    # removing batch dimension
    feature_map = feature_map.squeeze(0)

    # getting the number of channels in the feature map
    num_channels = feature_map.shape[0]

    num_to_show = min(4, num_channels)

    # creating the figure
    fig, axes = plt.subplots(1, 4, figsize=(5, 2))
    fig.suptitle(f"{layer_name} Feature Maps for Image 5")

    # plotting
    for channel in range(num_to_show):
        ax = axes[channel]
        ax.imshow(feature_map[channel].cpu().numpy(), cmap="viridis")
        ax.axis("off")
        ax.set_title(f"Channel {channel}")

    for i in range(num_to_show, 4):
        fig.delaxes(axes[i])

    plt.savefig(f"plots/feature_map_{layer_name}.png")
    plt.close()
```

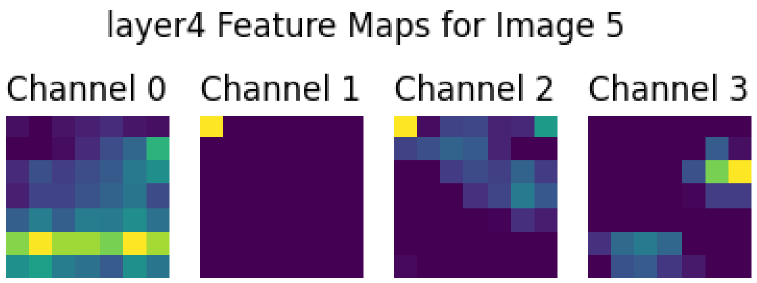
d)



From this example, we notice that layer1 captures finer edges and textures. The activations here are more evenly distributed, so they respond to smaller variations such as noice and fine details.



As we move deeper into the network (layer3, layer4) we notice that the feature maps become more sparse. The activations are now concentrated in specific regions, and they are most likely corresponding to high level structure or objects.



Finally we are left with low-resolution maps suggesting that the network has abstracted itself away from the original details, with the “heatmaps” suggesting in which part of the image the network has captured most information.

e), f)

[11]:
```python
from FeatureAnalysis import SparsityAnalyzer

module_names = ['layer1.0.relu', 'layer2.0.relu', 'layer3.0.relu', 'layer4.0.
 ↪relu', 'relu']
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=True)
analyzer = SparsityAnalyzer(model, module_names)
feature_stats = analyzer.analyze_activations(test_loader)
analyzer.print_statistics()
analyzer.cleanup()
```

```
Processed 1 images

Processed 200 images
Module layer1.0.relu: Average non-positive values: 43.89%, Count: 200
Module layer2.0.relu: Average non-positive values: 59.47%, Count: 200
Module layer3.0.relu: Average non-positive values: 64.00%, Count: 200
Module layer4.0.relu: Average non-positive values: 78.50%, Count: 200
Module relu: Average non-positive values: 33.16%, Count: 200
```

These results show that the layers are progressively sparse the deeper we go. This reflects what we see in the visualization of the feature maps: while early layers retain a mix of positive and zero activations, later layers are more sparse, suggesting that the network filters out irrelevant details and focuses on high-level structures. The final ReLU activation shows a lower percentage of non-positive values, though. This is because it needs to retain enough information for the classification: the last activation layer enhances key features so that the distinction between the classes can make sense.

A note on my setup

I run on my gaming PC with Nvidia RTX4090 GPU, which is quite optimal for ML tasks. The system runs Windows with WSL, allowing me to use a Linux environment for the development over SSH from wherever I am situated. I have set this up with VS Code's SSH-extension, and installed all necessary CUDA and NVIDIA drivers. This setup makes me able to train on my home computer from everywhere, freeing up system resources on my laptop.