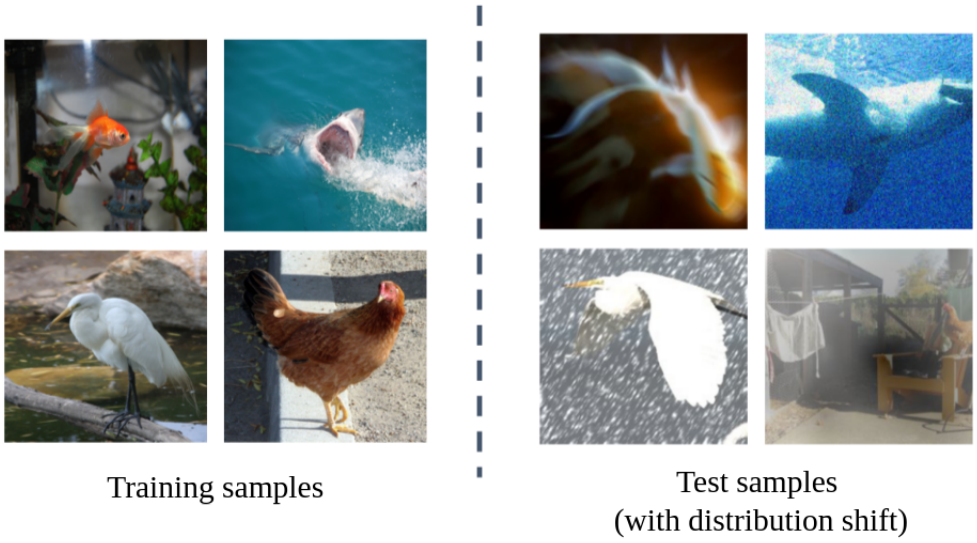

Figure 1: Examples of distribution shift

1. PyTorch Installation

If you haven't already: In your miniconda environment, install PyTorch and TorchVision from here, the exact command to setup with GPU support depends on your system. On the pool computers use this command:
`conda install -y pytorch torchvision pytorch-cuda=11.8 -c pytorch -c nvidia`
Afterwards check if your GPU is detected (the command should print True):
`python -c 'import torch; print(torch.cuda.is_available())'`
In case you get the error `chardet not found`, run `pip install chardet`
See learn the basics and other tutorials to get started with PyTorch.

2. Accessing the data

The data folder for this exercise can be found on server of the technical faculty in
`/project/cv-ws2425/lmb/data`
To run locally with a different folder use
`--data_dir /path/to/data`

3. Batch Normalization and Distribution shift

CNNs are known to perform well when the test and training data are sampled from the same distribution (i.e are *i.i.d*). However, in many real world scenarios this assumption does not hold. Images might differ from the training data if lighting conditions change or if dirt particles accumulate on the camera. In medical imaging, MRI scans will differ from the training data if different acquisition systems are being used. While human vision is robust to such distribution shifts, modern machine vision models are not.

In computer vision, Batch Normalization (BN) is a popular technique to improve convergence of CNNs. BN estimates the statistics of activations for the training dataset and uses them to normalize intermediate activations in the network. See torch.nn.BatchNorm2d for more details. By design, activation statistics obtained during training time do not reflect the statistics of the test distribution (Eg: when testing in out-of-distribution settings like corrupted images). Adapting BN statistics at test time is a popular technique for domain adaptation to new distributions, which is the goal of this exercise. Some examples of distribution

shifts are shown in Figure 1.

In this exercise we will be using a pre-trained model and implement a method for updating BN statistics. The code for running evaluations is `run_resnet.py` where you have to implement the function `update_bn_params`. This method takes the model, the data loader and command line arguments. The number of iterations can be set using the argument `num_bn_updates`. The script already has options to load different datasets with distribution shifts (common corruptions). The available corruptions are: `'defocus_blur','glass_blur', 'motion_blur', 'zoom_blur', 'snow', 'frost', 'fog', 'brightness'`. You can also control the corruption severity using the `severity` flag (range is from 1 to 5).

Todo: Run `run_resnet.py --clean_eval` to test the network on clean images and report the performance.

Todo: Run `run_resnet.py --evaluate` To check the impact of all corruption types at different levels of severity.

This can take quite a while to complete so you can also run e.g `run_resnet.py --corruption "frost" --severity 2` to test individual corruptions at a chosen severity.

Todo: Fill in the Todo block around the `update_bn_params` function in order to briefly train on the validation set, so the batch norm can be updated before validation.

Todo: Now, run `run_resnet.py --corruption "frost" --severity 2 --apply_bn` and compare your results to the unadapted version.

Todo: Set the number of batch norm updates as a the hyper parameter `--num_bn_updates` and play around with it to see how it affects performance.

Todo: Keep the number of updates fixed to 50 and consider the following batch sizes: 1, 4, 16, 64. Explain your observations.

This assignment will be introduced on 18.11. 16:00 CET. The solution will be released and discussed in the following session.