1. Installation

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. Convolutions using Python & Numpy

Convolutions are instrumental when creating convolutional neural networks or just for general image processing filters such as blurring, sharpening, edge detection, and many more. They are based on the idea of using a kernel and iterating through an input image to create an output image. They also make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

In this section, we will be implementing a 2D Convolution and then applying an edge detection kernel to an image using the 2D Convolution:

Todo: Compute the output shape as a function of generic values of stride, padding and kernel size.

Todo: Reshape the input by applying equal padding to all sides of the input.

Todo: Add the appropriate conditions to apply the convolution at specified strides only.

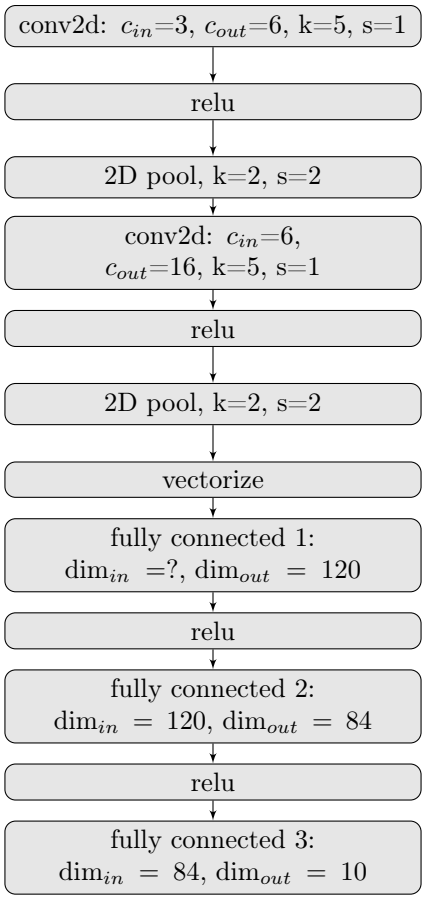
Todo: Compute the output value and assign it to the corresponding output location

Todo: Call the 2D convolution and visually check the output of the edge detector.

3. Convolutional neural network for image classification:

overfitting, visualization and data augmentation

Your first task will be to implement the small convolutional network LeNet. LeNet is composed of a few basic functions: 2D convolution, an activation function, 2D max pooling and fully connected layers. As activation function we will use the Rectified Linear Unit (ReLU). LeNet is defined by



, where c denotes channels, k the kernel size, s the stride and dim the input or output dimension.

For this exercise you can control parameters and change the control flow by setting flags with argparser. For some tasks it might be useful to define your own arguments. Argparse is a very basic python tool, which allows you to pass arguments in the command line. It is used in basically all code bases. If this is new to you familiarize yourself with argparse https://docs.python.org/3/library/argparse.html.

Todo: Please fill out the code skeleton in `ex03/lib/lenet_model.py`. Determine the input dimension of the first fully connected layer yourself. For this task you are allowed to use:

- `torch.nn.Conv2d`
- `torch.nn.Linear`
- `torch.nn.functional.relu`
- `torch.nn.functional.max_pool2d`
- `torch.Tensor.view`
- `torch.nn.Sequential`

Todo: Train your network with the given training code. Extend the training code with a validation and test step. Make sure to not train on your test and validation step and that you do not update any parameters of the network. This is in particular important when using a network with batch norm operation. To this end, read about `torch.no_grad()` and `model.eval()`. Log training and validation accuracy in a .txt file. Why is the test accuracy much worse than the training accuracy? What happens here?

Todo: You might have realized that logging the values is a good way to roughly follow the training, however, it quickly becomes confusing. In particular if you want to compare multiple training runs with different parameters. A great tool to monitor training statistics of multiple models is tensorboard. Extend the training, test and validation functions to log the loss and accuracy in tensorboard. To set it up you can find infomation online, f.e. here https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html.

Todo: To prevent this degradation in test accuracy one way is to stop training as soon as the validation accuracy starts to drop. Besides that, many other methods exist to prevent overfitting, some of which even improve the final test accuracy. Of these methods data augmentation is probably the most important one in computer vision. Common augmentations are cropping, scaling, flipping, rotation and color jitter. As you can see in the function `get_transforms` we only normalize the input images. To further increase the accuracy implement the two basic augmentations “flipping" “random resize crop". To this end fill out the code skeletons in `augmentations.py`.

- Flipping: Flips the image along the x-axis with a probability of 0.5.
- Random resize crop: Crop a part of the image and resize it to the expected input size.

PIL and Pytorch both implement these functions, but you should implement it yourself. You are allowed to use a resizing function from any libraray.

Todo: Overfitting should now be a problem of the past. Note that you can boost accuracy further by using more sophisticated augmentations, combining them and choosing better parameters. Look into https://pytorch.org/vision/stable/transforms.html and play around with more augmentations. Try to find a better set of parameters.

This assignment will be introduced on 04.11. 16:00 CET. The solution will be released and discussed in the following session.