General hints:

- Your code should work with *Python 3.8*.
- We suggest to adhere to the PEP8 style guide and use line lengths of up to 120.
- `for` loops can be slow in Python, use vectorized `numpy` operations wherever possible (see assignment 1 for an example).

How to run the exercise and tests:

- See the setup instructions downloadable from our website for installation details.
- We always assume you run commands in the *root folder* of the exercise.
- If you're using miniconda, don't forget to activate your environment with `conda activate mycvenv`
- Install the required packages with `pip install -r requirements.txt`
- Python files in the *root folder* of the repository contain the scripts to run the code.
- Some exercises contain unittests in folder `tests/` to check your solution. Run `python -m pytest .` to run all tests.
- To check your solution for the correct code style, run `pycodestyle -max-line-length 120 .`

In this exercise you will create a small convolutional neural network and train it on the CIFAR-10 dataset. The task is to classify each image into one of the 10 classes.

1. Installation

We assume that you have installed python 3.8 using miniconda as described in the setup instructions in the `setup.pdf` on the website of the Computer Vision lecture. You should know how to navigate and run commands in the command prompt.

In your miniconda environment, install the latest versions of PyTorch and TorchVision as described on PyTorch's website. **Optional**: To use a GPU install PyTorch with the compute platform cuda. Use the highest available cuda version and downgrade if necessary, this depends on your GPU. Make sure your GPU drivers are up to date. For this exercise using a CPU is fast enough.

2. Tutorials

If you have problems working through this exercise we suggest to check out some of the tutorials for PyTorch, learn the basics is a good tutorial to get started.

3. Parsing commandline arguments

We will use the script `run_cifar.py` for all experiments in this exercise. It uses argparse to create a user interface in the command line.

Todo: Run `run_cifar.py --help` to see which arguments are available.

4. Loading the data

To train our model we will need some data. We will use the CIFAR10 dataset which is provided in the TorchVision package. To do this, we will need to create datasets and wrap them into dataloaders as described here.

In file `lib/cifar_dataset.py` we have already defined the function `create_cifar_datasets` to create the train and test set.

Todo: Fill in the function `create_dataloader` in the same file to create a dataloader given a dataset. **Todo:** Run `run_cifar.py --test_dataloader` to see if your dataloader works.

Note: If you get an error during downloading the dataset, e.g. "ssl: certificate verify failed", you can download it manually here, create the folder `data/` and move the downloaded file `cifar-10-python.tar.gz` into that folder.

Now we have input data and labels to train our model. It is important to understand the shapes you are working with:

The image data is a 4D tensor of shape (batch_size=64, input_channels=3, input_height=32, input_width=32) with 3 channels for RGB and 32x32 pixel resolution. The class labels are a 1D tensor of shape (batch_size=64).

5. **Creating the model**

Todo: Fill in the todo blocks in the file `lib/cifar_model.py` to create the model and define the forward pass. **Todo:** Run `run_cifar.py --test_model` to see if your model works.

Note: By default, the model and all tensors will run on the CPU. To use a GPU, we need to define our device and move the model and all data to the device. This is done in `run_cifar.py`: under `# set our device` we define the device as the first GPU if one is available, otherwise we use the CPU. After model creation we move the model to that device and we also move any data we use to that device.

6. **Creating the loss function**

Note: We will leave the 1st todo block about model loading for later.

Todo: Fill in the 2nd todo block in `run_cifar.py` to create the Cross-Entropy-Loss. Note that this loss expects so called "logits". Our model creates the logits of the prediction with the last linear layer. To transform the logits into actual probabilities (i.e. where the sum over all classes is 1) you could use the softmax function, however the Cross-Entropy-Loss calculates the softmax internally and therefore expects logits.

7. **Creating the SGD optimizer**

Todo: Fill in the 3rd todo block in `run_cifar.py` to create the Stochastic Gradient Descent optimizer. You can leave out the next block with the AdamW optimizer for now.

8. **Creating the training loop**

Training a model looks like this: We loop over a given number of epochs, where one epoch means one iteration over the entire training set. For each epoch, we loop over the training set one batch at a time and update the model parameters given the loss and the optimizer.

Todo: Fill in the 5th todo block in `run_cifar.py` to create the training loop.

9. **Creating the test loop**

To test, we simply iterate over every batch in the test set and calculate the average loss and accuracy. In this script we test once after every training epoch.

Todo: Fill in the 6th todo block in `run_cifar.py` to create the test loop.

10. **Run the training**

Todo: Now you can run the training with `run_cifar.py` and you should see the loss go down and validation accuracy improve each epoch. Don't worry about the `NotImplementedError` at the end, we will fix that later. After training is done you should get an accuracy of about 30%. This is at least better than the random baseline of 10%, so we know that our model has learned something.

11. **Saving and loading**

Todo: Fill in the last todo block in `run_cifar.py` to save the model after training.

Todo: Run the training again and confirm that a model file is created in the code folder.

Todo: Fill in the first todo block in `run_cifar.py` to load the model you just saved.

Todo: Run `run_cifar.py --load_model model_e10_sgd_f32_lr1.0e-03.pth --validate_only`. You should get the same results as at the end of the training.

12. **Final remarks**

We have skipped some concepts here to keep the exercise short:

Model initialization: When you create the model, it's weights will be initialized randomly, usually close to zero. In this exercise we have left it up to PyTorch to initialize the weights. The conv2d and linear layers are initialized with the Kaiming Uniform method as described in the torch.nn.init documentation. In general, it can be worth to try out other kinds of initialization.

In the optimization loop, we use `model.train()` and `model.eval()` to set the model for training and evaluation mode respectively. This is because we want some layers to behave differently during evaluation: During training, BatchNorm2d will normalize the input batch and learn it's mean and variance, while during evaluation, BatchNorm2d will use the learned mean and variance to normalize the data. Dropout (which we did not use during this exercise) will be disabled during evaluation.

We created the dataset using `torchvision.datasets` while in practice you will usually have to create your own custom dataset.

To work with the PyTorch tensors that are output by the model you usually need to detach them from the optimization process, move them to cpu and convert them to numpy arrays or python variables. We used `tensor.item()` to convert single elements like the loss to a python float. To convert a tensor to a numpy array, e.g. to look at the model predictions, we would use `tensor.detach().cpu().numpy()`.

13. **Bonus parts**

1) Parallel data loading

   With the `--num_workers` parameter we can use several CPU workers to load the data.

   Todo: Try setting it to a value greater than one and see if the training speed increases.

2) The AdamW optimizer

   AdamW uses adaptive learning rates and is the defacto standard optimizer used in deep learning.

   Todo: Implement the 4th todo block to create the AdamW optimizer.

   Todo: Run the training with the AdamW optimizer and reduced learning rate, as usually AdamW needs a lower learning rate than SGD: `run_cifar.py --optimizer adamw --learning_rate 3e-4`

3) Accuracy on the training set

   It can be interesting to compare accuracy on the training and test set.

   Todo: Calculate the average accuracy on the training set in the training loop and print it.

4) Hyperparameter Optimization

   Todo: Run the training several times and vary the batch size, learning rate, choice of optimizer, number of epochs and number of convolutional filters. How much accuracy can you achieve?

5) Neural Architecture

   The model we created is very simple and the accuracy will probably not be very good even after trying out different hyperparameters, since the architecture is just not very good.

   Todo: Try to improve the model architecture. You could for example add more convolutional layers and average / max pooling layers.

   Todo: Architectures that have been proven to be strong are e.g. ResNets. You could use a pretrained ResNet-18

from PyTorch's model hub here and train it with the optimization loop we defined. This is called finetuning on the target CIFAR10 dataset with a model that has been pretrained on a different dataset.

Note: In this case the model will output a distribution over the 1000 ImageNet classes, while you are finetuning on 10 classes from CIFAR10. This works reasonably well, but the best practice here is to remove the last layer from the ResNet-18 and add a new linear layer with the correct number of output classes.

This assignment will be introduced on 03.11. 12:00-13:00. The solution will be discussed one week later.