Computer Vision - Exercise 4
Object Recognition

In this exercise, we will implement:

- A bag-of-words image classifier that decides whether a test image contains a car (back view) or not.
- A CNN-based image classification network on CIFAR-10 dataset for multi-class image classification.

The data is contained in the `data` folder. Your task is to complete missing parts of the provided code as described in this handout and in the code itself.

1 Environment Setup

For setting up the Python environment, we will be using miniconda. For the best and easiest experience, we recommend using a Linux distribution (e.g., Ubuntu). However, miniconda is also compatible with Windows and Mac. Please install the latest miniconda version for your OS from the following link https://docs.conda.io/en/latest/miniconda.html. Once the miniconda is installed, you can run the following commands from the root of the code directory to install and activate the environment:

```
conda env create -f env.yml

conda activate ex4
```

Optional: If you are using a CUDA-compatible machine and want to accelerate model training and inference, you can alternatively set up the environment by:

```
conda env create -f env_cuda.yml

conda activate ex4_cuda
```

2 Bag-of-words Classifier (60%)

You need to fill the missing code in `bow_main.py` script.

2.1 Local Feature Extraction (20%)

2.1.1 Feature detection - feature points on a grid (5%)

Implement a very simple local feature point detector: points on a grid. Within the function `grid_points`, compute a regular grid that fits the given input image, leaving a border of 8 pixels in each image dimension. The parameters `nPointsX` $= 10$ and `nPointsY` $= 10$ define the granularity of the grid in the x and y dimensions. The function shall return `nPointsX` $\cdot$ `nPointsY` 2D grid points. The function to be written takes the following form:

```
vPoints = grid_points(img,nPointsX,nPointsY,border)
```

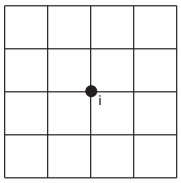

Figure 1: Grid point i and surrounding cells.

2.1.2 Feature description - histogram of oriented gradients (15%)

Next, we describe each feature (grid point) by a local descriptor. We will use the well known histogram of oriented gradients (HOG) descriptor. Implement the function `descriptors_hog` which should compute one 128-dimensional descriptor for each of the N 2-dimensional grid points (contained in the $N \times 2$ input argument `vPoints`). The descriptor is defined over a 4×4 set of cells placed around the grid point i (see Fig. 1).

For each cell (containing `cellWidth` $\times$ `cellHeight` pixels, which we will choose to be `cellWidth` $=$ `cellHeight` $=$ 4), create an 8-bin histogram over the orientation of the gradient at each pixel within the cell. Then concatenate the 16 resulting histograms (one for each cell) to produce a 128 dimensional vector for every grid point. Finally, the function should return a $N \times 128$ dimensional feature matrix. The function to be written takes the following form:

```
descriptors = descriptors_hog(img,vPoints,cellWidth,cellHeight)
```

2.2 Codebook construction (15%)

In order to capture the appearance variability within an object category, we first need to construct a visual vocabulary (or *appearance codebook*). For this we cluster the large set of all local descriptors from the training images into a small number of visual words (which form the entries of the codebook). For this, you will use the K-means clustering algorithm. This algorithm performs the following basic steps:

- Initialize each of the `k` cluster centers with the position of a randomly chosen feature from the set of input features `vFeatures`.
- Repeat for `numiter` iterations.
  - Assign each input point to the closest cluster center.
  - Shift the position of each cluster center to the mean of its assigned points.
- Return the `k` cluster centers.

The K-Means clustering algorithm is already efficiently implemented in Python `sklearn` library. Fill the function `create_codebook` which takes as input the name of a directory, loads each image in it in turn, computes grid points for it, extracts descriptors around each grid point, and clusters all the descriptors with K-Means. It then returns the found cluster centers. Make sure to use all descriptors from all training images in the directory. The function to be written takes the following form:

```
vCenters = create_codebook(nameDir,k,numiter)
```

2.3 Bag-of-words Vector Encoding (15%)

Using the appearance codebook created at the previous step we will represent each image as a histogram of visual words. This representation, also known as the **bag-of-words** representation, records which visual words are present in the image.

2.3.1 Bag-of-Words histogram (10%)

Fill the function `bow_histogram` that computes a bag-of-words histogram corresponding to a given image. The function should take as input a set of descriptors extracted from one image and the codebook of cluster centers. To compute the histogram to be returned, assign the descriptors to the cluster centers and count how many descriptors are assigned to each cluster (i.e. to each 'visual word'). The function to be written takes the following form:

```
bowHist = bow_histogram(vFeatures, vCenters)
```

2.3.2 Processing a directory with training examples (5%)

Fill the function `create_bow_histograms` that reads in all training examples (images) from a given directory and computes a bag-of-words histogram for each. The output should be a matrix having the number of rows equal to the number of training examples and number of columns equal to the size of the codebook (number of cluster centers). The function to be written takes the following form:

```
vBoW = create_bow_histograms(nameDir,vCenters)
```

Using this function we process all positive training examples from the directory **cars-training-pos** and all negative training examples from the directory **cars-training-neg**. We collect the resulting histograms in the rows of the matrices `vBoWPos` and `vBoWNeg`.

2.4 Nearest Neighbor Classification (10%)

We build a bag-of-words image classifier using the nearest neighbor principle. For classifying a new test image, we compute its bag-of-words histogram, and assign it to the category of its nearest-neighbor training histogram. Fill the function `bow_recognition_nearest` that compares the bag-of-words histogram of a test image to the positive and negative training examples and returns the label 1 (car) or 0 (no car), based on the label of the nearest-neighbour. The function to be written takes the following form:

```
label = bow_recognition_nearest(histogram,vBoWPos,vBoWNeg)
```

3 CNN-based Classifier (40%)

Here we implement a simplified version of the VGG image classification network (`https://arxiv.org/pdf/1409.1556.pdf`) on CIFAR-10 dataset (`https://www.cs.toronto.edu/~kriz/cifar.html`). CIFAR-10 dataset includes 10 image classes, with 50000 training images, from which we split 5000 images for validation, and 10000 testing images, of resolution 32×32.

3.1 A Simplified version of VGG Network (20%)

The code skeleton of a simplified version of VGG Network can be found in `models/vgg_simplified.py`. You will need to implement the network architectures in the initial function and the `forward()` function of class `Vgg`. The simplified VGG network architecture is shown in Tab. 3.1.

Block Name	Layers	Output Size
conv_block1	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 64, 16, 16]
conv_block2	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 128, 8, 8]
conv_block3	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 256, 4, 4]
conv_block4	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 512, 2, 2]
conv_block5	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 512, 1, 1]
classifier	Linear+ReLU+Dropout+Linear	[bs, 10]

3.2 Training and Testing (20%)

3.2.1 Training (10%)

The dataloader, training, and testing scripts are already implemented, and it's highly recommended to read them carefully to understand the full pipeline. To train the model, use the following command:

```
python train_cifar10_vgg.py --save_dir=runs
```

You can tune the parameters in `args` in `train_cifar10_vgg.py` to adapt the code to your setup. The training script will save the trained models, and training / validation logs by `tensorboard`. You can monitor how the model learns using `tensorboard` by running the following command:

```
tensorboard --logdir PATH_TO_LOG_FOLDER
```

With the default parameters, the training takes around 4 hours in a laptop with i5 Core CPU and 16GB memory for 30 epochs. It should be sufficient to run 30-40 epochs but you are encouraged to train for longer time to investigate how the loss curve changes, and you can use either CPU or GPU for training. Moreover, you can also reduce the number of training epochs (e.g., to 10 epochs) if the training time is too long on your machine.

You need to include a screenshot from `tensorboard` of your training loss and validation accuracy curve in the hand-in report. You also need to hand in the `runs/xxxxx` folder generated by the training script, which includes:

- tensorboard log file named `events.out....`
- your trained model named `last_model.pkl`
- your parameter setup named `params.json`
- a log file named `run_XXXX_XX_xx_xx_xx_xx.log`

3.2.2 Testing (10%)

To test the model performance with your trained model, use the following command:

```
python test_cifar10_vgg.py
```

Please specify the path to your trained model and other parameters in the script by setting the default argument values. Please report your test accuracy on CIFAR-10 test set in the hand-in report.

For this part, you will get full 10 pts if your test accuracy exceeds 0.6 and include all required documents / materials. The points will be scaled linearly between 0-10 if your final test accuracy is less than 0.6. Note that if the test script cannot load your model, then you will get 0 (out of 10) points for this subtask.

For reference, a sufficiently trained model with the default parameters can achieve a test accuracy of around 0.8. Achieving a test accuracy of 0.6 normally requires less than 10 epochs of training. You could choose to terminate the training before the model accuracy converges if training is too slow on your machine.

4 Hand-in

Please hand in 1) all the code (excluding the `data` folder); 2) `runs` folder generated by `train_cifar10_vgg.py` including training logs and your trained model; 3) a short pdf report to explain your implementations and your results. For the bag-of-words classifier, please include the logging of running `bow_main.py`. For the CNN-based classifier, please include the logging of running `test_cifar10_vgg.py` and the `tensorboard` screenshots of training and validation curves.

Please ensure that your `test_cifar10_vgg.py` can be directly run by the command `python test_cifar10_vgg.py` to load your trained model and test accuracy on CIFAR-10.

Please zip all files (excluding the `data` folder) into one single file for submission.