# Assignment 4 - Object Recognition

**Federica Bruni** (fbruni@student.ethz.ch) 9 November 2023

## 1 Overview

This assignment included the implementation of two distinct image classification tasks.

Firstly, I constructed a Bag-of-Words (BoW) image classifier designed to discern whether a given test image features a car, specifically from its back view.

Secondly, I created of a Convolutional Neural Network (CNN)-based image classification network. This network will be trained on the CIFAR-10 dataset. My goal here is to develop a model that excels at assigning each image accurately to one of these predefined classes.

## 2 Bag-of-words Classifier

### 2.1 Feature detection - feature points on a grid

The first task required implementing a local feature point detector that creates a regular grid to cover an input image while leaving an 8-pixel border on each dimension.

The goal was to generate grid points with a specific granularity in the x and y dimensions, as defined by the parameters nPointsX and nPointsY.

To accomplish this, the function named 'grid_points' that takes an input grayscale image and these parameters and produces a 2D grid of point coordinates.

At the beginning the height (h) and width (w) of the input grayscale image are extracted.

```
h, w = img.shape
```

The next step involves calculating the x and y coordinates for the grid points. This calculation excludes the border, which is specified by the border parameter. To distribute the grid points evenly, we use the np.linspace function to create a range of coordinates along the x and y dimensions.

```
x_coords = np.linspace(border, w - border, nPointsX)
y_coords = np.linspace(border, h - border, nPointsY)
```

Then a meshgrid of these x and y coordinates is created using np.meshgrid. This essentially creates two matrices (x_mesh and y_mesh) that represent all possible combinations of x and y coordinates within the specified ranges.

```
x_mesh, y_mesh = np.meshgrid(x_coords, y_coords)
```

Finally, I stack the x and y coordinates from the meshgrid matrices into a single 2D grid of point coordinates.

```
vPoints = np.column_stack((x_mesh.ravel(), y_mesh.ravel()))
```

### 2.2 Feature description - histogram of oriented gradients

The next step was to characterize each feature grid point using a local descriptor. The Histogram of Oriented Gradients (HOG) descriptor is utilized.

The task involves the implementation of a function, "descriptors_hog,"responsible for generating 128-dimensional descriptors for each of the N 2-dimensional grid points, provided as input in the form of an $N \times 2$ matrix named "vPoints."

The goal is to copute the gradients of the image within a 4x4 cell around a given grid point. This is done by computing the x and y gradients within the specified cell region.

At the beginning the gradients in the x and y direction within a specific cell are calculated.

```
cell_gradients_x = grad_x[start_y:end_y, start_x:end_x]
cell_gradients_y = grad_y[start_y:end_y, start_x:end_x]
```

The next step involves calculating the angles of the gradients within the cell.

```
cell_angles = np.arctan2(cell_gradients_y, cell_gradients_x)
```

Then a histogram is computed for the cell_angles array. The np.histogram function calculates a histogram of values, in this case, the angles of gradients.

```
histogram, _ = np.histogram(cell_angles, bins=nBins, range=(-np.pi, np.pi))
```

Finally,the histogram values are accumulated in the desk variable.

```
desc += histogram
```

## 2.3 Codebook construction

To capture the diversity in the appearance of objects within a particular category, the next step is to create a visual vocabulary or an appearance codebook.This vocabulary is constructed by clustering a large collection of local descriptors obtained from training images into a compact set of visual words, which form the entries of the codebook. In this process, I utilize the K-means clustering algorithm, which follows these fundamental steps:

Collect local feature points by calling the 'grid_points' function, determining where these points should be in the image.

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

Compute a descriptor for each local feature point using the 'descriptors_hog' function.

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

Append these descriptors to the 'vFeatures' list

```
vFeatures.append(img_descriptors)
```

## 2.4 Bag-of-Words histogram

Utilizing the appearance codebook created in the previous step, each image is represented using a histogram of visual words. This representation keeps track of which visual words from the codebook are present in a given image. To accomplish this, I implemented the 'bow_histogram' function, which computes a bag-of-words histogram specific to an input image.

At the beginning the number of cluster centers N and the number of features M is determined and the histo array is initialized.

```
N = vCenters.shape[0]
M, _ = vFeatures.shape
histo = np.zeros(N)
```

For each descriptor in the feature set it calculates the Euclidean distance to all cluster centers.

```
distances = np.linalg.norm(vFeatures[i] - vCenters, axis=1)
```

Identify the index of the closest cluster center by finding the cluster with the minimum distance.

```
closest_cluster_index = np.argmin(distances)
```

Finally, increment the count of the corresponding cluster center in the 'histo' array.

```
histo[closest_cluster_index] += 1
```

## 2.5 Processing a directory with training examples

I have implemented the 'create_bow_histograms' function to process all the training examples (images) within a given directory. This function reads in images and calculates a bag-of-words (BoW) histogram for each of them. The output is a matrix, where the number of rows corresponds to the number of training examples, and the number of columns is equal to the size of the codebook. Here's an explanation of my code:

r each image in the specified directory, I start by collecting local feature points by using the 'grid_points' function.

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

For each local feature point, I compute a descriptor using the histogram of oriented gradients (HOG) method.

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

Once I have descriptors for all local feature points in the image, I use the 'bow_histogram' function to calculate the BoW histogram for that image.

```
img_histogram = bow_histogram(img_descriptors, vCenters)
```

I append the image's BoW histogram to a list, 'vBoW', which collects the histograms for all training examples processed in the directory.

```
vBoW.append(img_histogram)
```

### 2.6 Nearest Neighbor Classification

Finally to classify a new test image, the bag-of-words histogram is computed and assign it to the category of the nearest-neighbor training histogram. Here's an explanation of the 'bow_nearest' function:

I calculate the distances (or similarities) between the bag-of-words histogram of the test image, represented as 'histogram,' and the positive ('vBoWPos') and negative ('vBoWNeg') training examples' histograms.

```
DistPos = np.min(np.linalg.norm(vBoWPos - histogram, axis=1))
DistNeg = np.min(np.linalg.norm(vBoWNeg - histogram, axis=1))
```

Then the final classification decision is made based on the nearest-neighbor principle. If 'DistPos' (distance to the positive training set) is smaller than 'DistNeg' (distance to the negative training set), it's labeled as 1 (car), indicating that it is closest to a positive training example. Otherwise, if 'DistNeg' is smaller, it is labelled as 0 (no car), indicating that it is closest to a negative training example.

## 3 CNN-based Classifier

### 3.1 A Simplified version of VGG Network

In the implementation of the simplified VGG network, I created a convolutional neural network (CNN) model with a structure inspired by the VGG architecture. The goal is to construct a network capable of classifying images into one of ten classes, making it suitable for the CIFAR-10 dataset. Here's a breakdown of how this VGG-like network is implemented:

Five convolutional blocks (conv_block1 to conv_block5) are created using nn.Sequential. Each block consists of: A 2D convolutional layer with 3 input channels, a specified number of output channels (e.g., 64 in conv_block1), a kernel size of 3x3, and padding of 1. A ReLU activation function applied in-place. A 2D max-pooling layer with a kernel size of 2x2 and a stride of 2, which reduces the spatial dimensions of the feature maps.

```
self.conv_block1 = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2))

self.conv_block2 = nn.Sequential(
    nn.Conv2d(64, 128, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2))

self.conv_block3 = nn.Sequential(
    nn.Conv2d(128, 256, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2))

self.conv_block4 = nn.Sequential(
    nn.Conv2d(256, 512, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2))
```

```
self.conv_block5 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2))
```

The classifier is implemented as a separate nn.Sequential block. It starts with a fully connected (linear) layer with 512 input features and a number of output features specified by the fc_layer parameter. A ReLU activation function is applied to the output of the first linear layer. A dropout layer with a dropout rate of 0.5 is included for regularization. The final fully connected layer maps the output to the number of classes defined by the classes parameter.

```
self.classifier = nn.Sequential(
    nn.Linear(512, fc_layer),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),
    nn.Linear(fc_layer, classes))
```

The forward method specifies the forward pass of the network. The input tensor x is passed through each of the convolutional blocks (conv_block1 to conv_block5), progressively reducing the spatial dimensions. The output from the last convolutional block is reshaped using .view to be compatible with the fully connected layers. The reshaped tensor is then passed through the classifier layers, producing the final class scores.

```
x = self.conv_block1(x)
x = self.conv_block2(x)
x = self.conv_block3(x)
x = self.conv_block4(x)
x = self.conv_block5(x)

x = x.view(x.size(0), -1)

score = self.classifier(x)
```

# 4 Results

## 4.1 Bag-of-words Classifier

The bag-of-words (BoW) classifier, as implemented, demonstrates high performance when evaluated on positive and negative test samples. Specifically, the classifier achieves an accuracy of 96% when classifying positive test samples, which correspond to images containing car back views, and an even higher accuracy of 98% when handling negative test samples, which represent images without cars.

These results highlight the effectiveness of the BoW image classification approach, where image representation is based on localized feature histograms and classification is determined using the nearest neighbor principle.

I report the bow_main.py log:

```
LOG:
    creating bow histograms (pos) ...
    creating bow histograms (neg) ...
    creating bow histograms for test set (pos) ...
    testing pos samples ...
    test pos sample accuracy: 0.9591836734693877
    creating bow histograms for test set (neg) ...
    testing neg samples ...
    test neg sample accuracy: 0.98
```

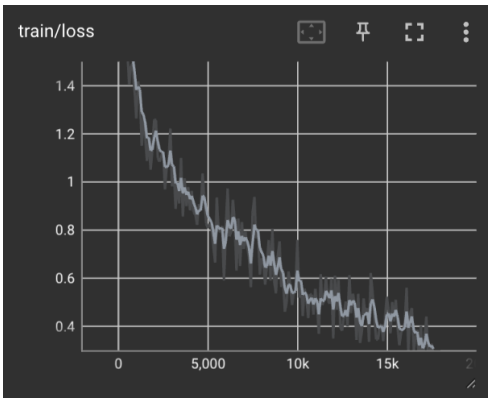
## 4.2 CNN-based Classifier

### 4.2.1 Training

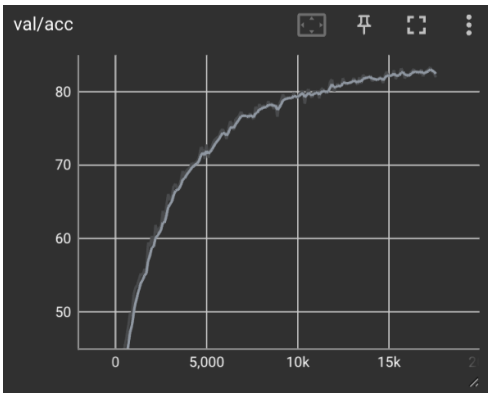
The Convolutional Neural Network (CNN) classifier, which was constructed and trained based on the architecture described earlier, achieved good performance in the image classification task. The best model obtained during training reaches an accuracy of 83.38% when classifying images, while maintaining a low loss value of 0.2843. These results reflect the efficacy of the CNN model in capturing and learning relevant features from the CIFAR-10 dataset, enabling it to categorize images into the ten different classes with a high degree of accuracy.

It is worth noting that during the training process, I observed that the loss value consistently decreased over time, a characteristic behavior indicative of the model's learning process. This implies that the model was progressively improving its ability to minimize classification errors and make more accurate predictions.

Additionally, the validation accuracy graph showed a remarkable trend of exponential increase. This trend highlights the model's growing proficiency in generalizing to unseen data, ultimately leading to improved accuracy.




Figuur 1: Training log




Figuur 2: Validation log

Figuur 3: Logs by tensorboard

#### 4.2.2 Testing

In the final phase the trained Convolutional Neural Network (CNN) model achieved an accuracy of 81.99% on the test dataset. This outcome underscores the model's robustness and its capacity to generalize well to unseen data.

```
LOG:
    [INFO] test set loaded, 10000 samples in total.
    79it [00:07, 10.08it/s]
    test accuracy: 81.99
```