*Prof. M. Pollefeyes*

# Filippo Ficarra: Lab 4 - Object Recognition

fficarra@student.ethz.ch, 22–938–062.

10/11/2023 - 02:07h

## 1 Introduction

The first part of the lab focuses on image recognition in a more classical way, using a method borrowed from NLP. The Bag of Words method for Image recognition it is adapted this way:

- Divide the images in subimages
- Create a codebook of explenative images
- Associate images to the one in the codebook to classify them

The second method uses a Deep Learning architecture to extract information and classify the images. This is done through the usage of a simplified version of the VGG architecture. This architecture is based on convolutional NN usually used for image recognition.

## 2 Bag of Words

### 2.1 Grid

The image is divided into a grid. The grid is created with the following steps:

- leave a border on all the edges. In this case border = 8
- take all the points that start from border + 1 to the image width - border and analogously for the height
- create an array of points from the step before

You can see this procedure in the following snippet:

Listing 1: Grid creation

```python
xs = np.linspace(border+1, image_width-border, nPointsX)
ys = np.linspace(border+1, image_height-border, nPointsY)

vPoints = np.array([(x, y) for x in xs for y in ys])
```

### 2.2 Histogram of Oriented gradients

In order to extract the feature we use the HOG algorithm. The variant used in this exercise doesn't use the magnitudes of the gradient but just their orientation.

For each grid point specified in vPoints, the algorithm computes a HOG descriptor using the following steps:

- It iterates over a 4x4 set of cells around the current grid point.

- For each cell, it calculates the gradient orientation and computes a hog. The orientation are taken in degrees.
- The histogram counts how many gradient orientations fall into each of the predefined bins (from -180 to 180 degrees, since arctan gives values in this range).

Listing 2: HOG

```
angle = np.arctan2(grad_y[start_y:end_y, start_x:end_x],
                         grad_x[start_y:end_y, start_x:end_x]) * 180 / np.pi
hist, _ = np.histogram(angle, bins=nBins, range=(-180, 180))
desc.append(hist)
```

## 2.3 Codebook

The book is generated gathering all the descriptors for all the sub images taken from the grid of all images and clusterizing them using the K-means cluster algorithm. The code was already implemented and the following code was added:

Listing 3: Codebook Generation

```
grid = grid_points(img, nPointsX, nPointsY, border)
descriptors = descriptors_hog(img, grid, cellWidth, cellHeight)
vFeatures.append(descriptors)
```

In the code I set a random_state = 42 to the K-means algorithm and numiteration = 300 (default). I will show in the result the effect of different ks.

## 2.4 Bag-of-Words histogram

In the function create_bow_histograms() we compute for the descriptors of all the images the histogram that relates each feature to a the centroids. The function relies on the function bow_histogram() function that computes the histogram of the features of an images with respect to the centroids.

## 2.5 Nearest Neighbour

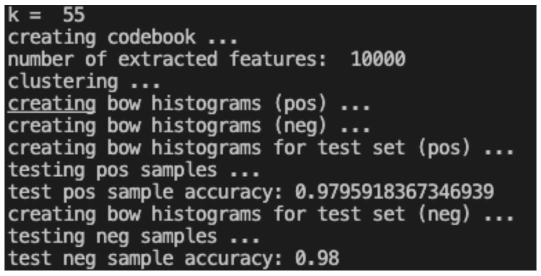
The classification is done using the findnn function to find both the distances of the positive BoW and the negative BoW. The smallest distance is the class we predict.

## 2.6 Results

The picture below show the trend of the posistive and negative samples when the number of centroids vary. It is useful to see that for little number of centroids like 5 the accuracy for the positive samples is very low, while increasing the number of centrioids we can be more accurate untill a certain amount of centroids. The more centroids the lower the accuracy tend to be. We can choose for example a value of k = 55, giving good results while keeping the complexity moderate.

You can find the log of the multiple k run into the file ”log_bow_main.txt”.

For the value k = 55 we have the following result:



```
k =  55
creating codebook ...
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
creating bow histograms (neg) ...
creating bow histograms for test set (pos) ...
testing pos samples ...
test pos sample accuracy: 0.9795918367346939
creating bow histograms for test set (neg) ...
testing neg samples ...
test neg sample accuracy: 0.98
```

Figure 1: BoW log

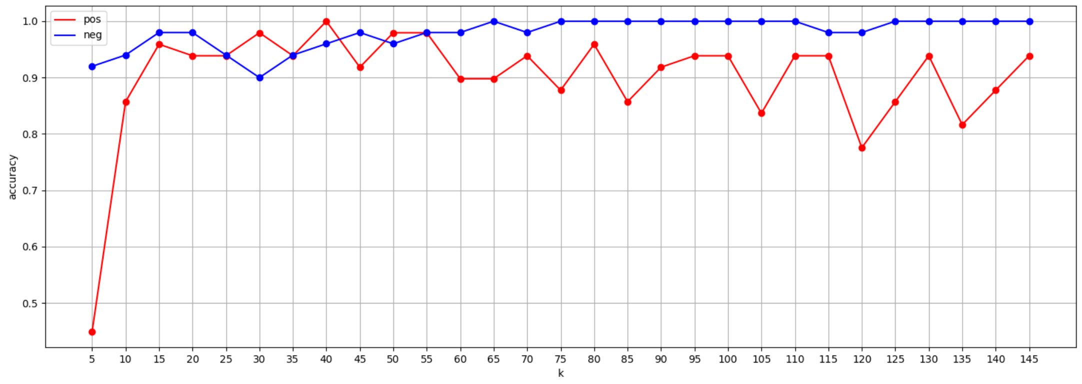
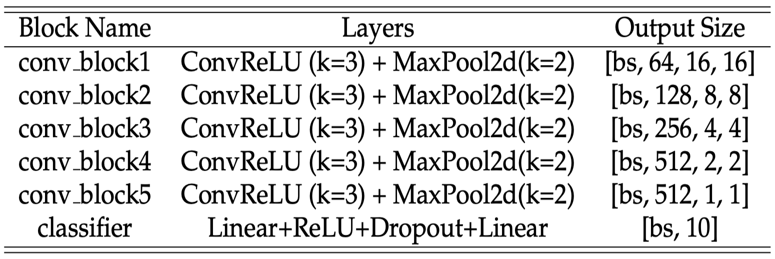


Figure 2: Test Accuracies for positive and negative samples

## 3 CNN-based Classifier

### 3.1 Training

The model is a simplified version of the VGG architecture. The architecture given was the following:



| 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] |

Figure 3: VGG simplified architecture

This has simply been implemented with torch like this:

Listing 4: VGG layers

```
self.conv_block1 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.conv_block2 = nn.Sequential(
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.conv_block3 = nn.Sequential(
    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.conv_block4 = nn.Sequential(
    nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.conv_block5 = nn.Sequential(
    nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.classifier = nn.Sequential(
    nn.Linear(512, self.fc_layer),
    nn.ReLU(),
    nn.Dropout2d(0.5),
    nn.Linear(self.fc_layer, self.classes)
)
```

The model was trained with the following parameters:

| **Argument** | **Value** | **Description** |
|---|---|---|
| batch_size | 128 | Batch size |
| log_step | 100 | How many steps to log once |
| val_step | 100 | Validation step |
| num_epoch | 50 | Maximum number of training epochs |
| fc_layer | 512 | Number of features in the first linear layer in VGG |
| lr | 0.0001 | Learning rate |

As shown in the table above the model was trained for 50 epochs with batch size 128.
Furthermore the training has been performed on Euler with 1 gpu RTX 4090.
Below we can find the graphs for the train losses in which we can see that the loss progressively drops from an initial value of 2 to a value around 1.3, while the validation accuracies pass from around 45% to a final value of 81.92%

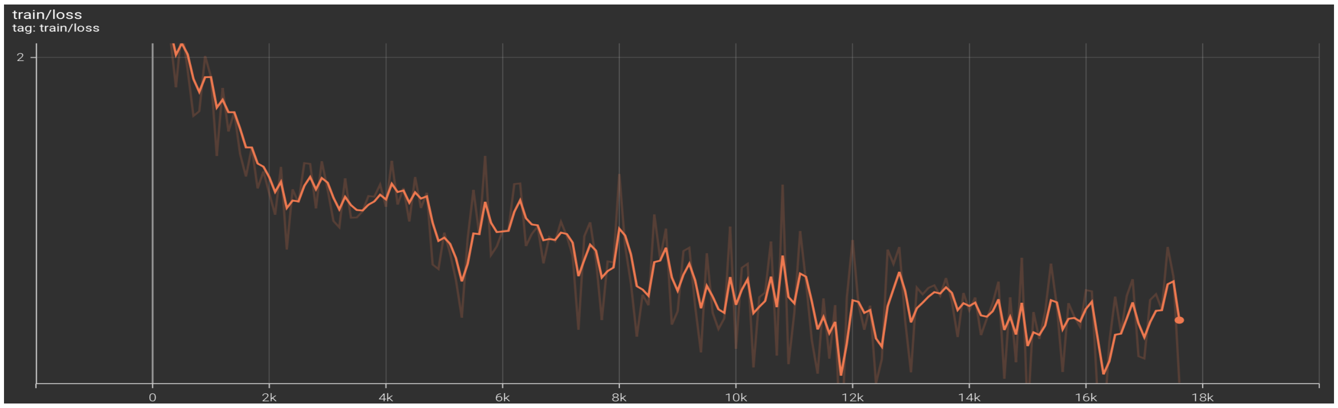


Figure 4: Train Losses

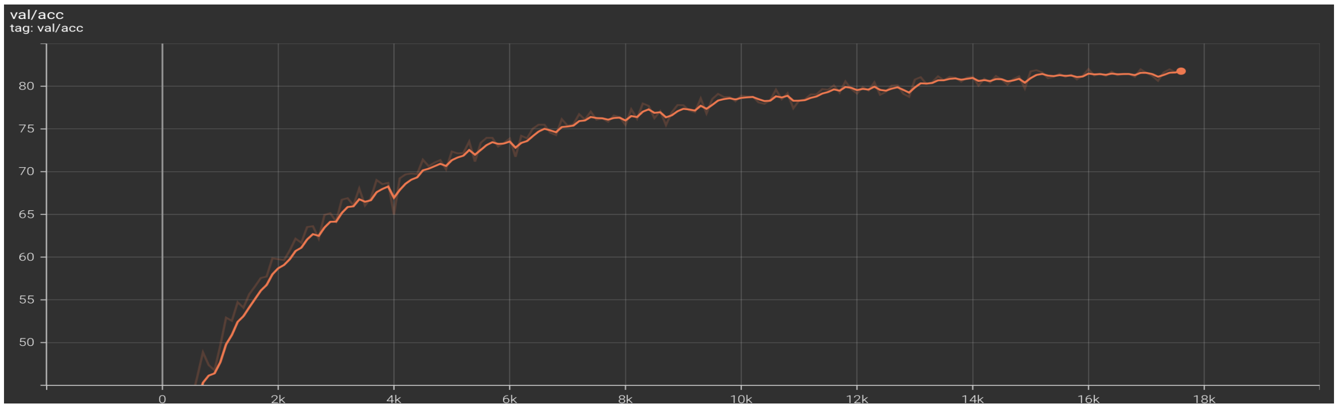
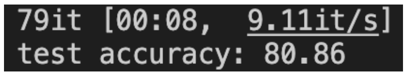


Figure 5: Validation Accuracies

## 3.2 Testing

The model after training with 50 epochs achieved a test accuracy of 80.86 on the CIFAR10 dataset. The test script was run locally on cpu for convenience and gave the following output:



```
79it [00:08,  9.11it/s]
test accuracy: 80.86
```