Lab assignment 4

ETH Zurich, Computer Science Department

Computer vision

Report

Object Recognition

Author
Emilia Szymańska, 22-945-547

November 2022

The objective of this lab assignment was to implement bag-of-words image classifier to decide whether a test image contains a car or not and to develop a CNN-based image classification network on CIFAR-10 dataset for multi-class image classification.

1 Bag-of-Words classifier

1.1 Implementation

1.1.1 Feature detection - feature points on a grid

First we have to define the $nPointsX \cdot nPointsY$ grid points. As we need to omit the borders in all directions, I run a function to get $nPointsX$ evenly spaced numbers over an interval from $border$ to $(width - border - 1)$. I perform a similar operation on Y axis case and then get an array of (x, y) coordinates created from the permutation of the previously obtained linspace vectors.

The code implementation of this part is presented below:

```
h, w = img.shape
x_vector = np.linspace(border, w-border-1, num = nPointsX, dtype=int)
y_vector = np.linspace(border, h-border-1, num = nPointsY, dtype=int)
grid = np.meshgrid(x_vector,y_vector)
grid_stacked = np.stack((grid[0], grid[1]), axis=-1)
vPoints = np.reshape(grid_stacked, (nPointsX*nPointsY, 2))
```

1.1.2 Feature description - histogram of oriented gradients

To calculate Histograms of Oriented Gradients (HOG), for each grid point we take 4x4 cell set, where each cell contains 4x4 pixels. We take gradients (in Y and X directions) corresponding to the currently inspected cell and calculate the magnitude $\sqrt{gradX^2 + gradY^2}$ and angle $arctan(gradY, gradX)$ (converted later to degrees) of the gradient resulting from Y and X gradients. As np.arctan2 function returns values from $-\Pi$ to Π I added 360^o to the negative angles to have the interval from 0 to 2Π. Then I calculated the histogram on the angles matrix with 8 bins over the $(0^o, 360^o)$ range, where the weights corresponded to the magnitude. Each set of histograms corresponding to one cell needed to be concatenated before adding to the resulting descriptor, which is then of size $(nPointsX \cdot nPointsY)x(4 \cdot 4 \cdot 8)$.

The code implementation of this part is presented below:

```
...
gradx_sliced = grad_x[start_y:end_y, start_x:end_x]
grady_sliced = grad_y[start_y:end_y, start_x:end_x]

x_squared = np.square(gradx_sliced, dtype=float)
y_squared = np.square(grady_sliced, dtype=float)
added = np.add(x_squared,y_squared, dtype=float)

mag = np.sqrt(added)
ang_neg = np.degrees(np.arctan2(grady_sliced, gradx_sliced))
ang = np.where(ang_neg < 0, ang_neg+360, ang_neg)

hist, binss = np.histogram(ang, bins=nBins, range=(0,360), weights=mag)
desc.append(hist)
...
```

1.1.3 Codebook construction

Codebook construction function required only adding the utilization of the previously defined functions (for grid points and HOG descriptor computation) and adding the result to the list of all features for all images.

The code implementation of this part is presented below:

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

1.1.4 Bag-of-Words histogram

To receive the Bag-of-Words activation histogram, it was necessary to first find nearest neighbors (from K-means centers) for each of the descriptors. Then I computed histogram based on vector of indices corresponding to the nearest neighbors for each descriptor.

The code implementation of this part is presented below:

```
...
idx, dist = findnn(vFeatures, vCenters)
histo, _ = np.histogram(idx, bins=vCenters.shape[0])
...
```

1.1.5 Processing a directory with training examples

Similarly to codebook construction, in create_bow_histograms function we just need to properly use the previously defined functions to obtain BoW histograms for each image (calculate grid points, extract descriptors, compute BoW histograms and add it to the set of all images' histograms).

The code implementation of this part is presented below:

```
...
vpoints = grid_points(img, nPointsX, nPointsY, border)
features = descriptors_hog(img, vpoints, cellWidth, cellHeight)
hist = bow_histogram(features, vCenters)
vBoW.append(hist)
...
```

1.2 Nearest Neighbor Classification

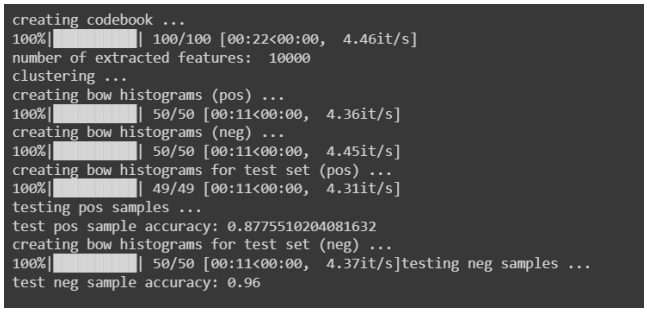
The last part required finding the nearest neighbors (from BoW for both positive and negative images) for the given histogram and on this basis return the corresponding image label.

The code implementation of this part is presented below:

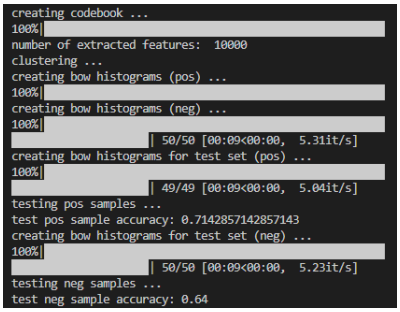
```
...
IdxPos, DistPos = findnn(histogram, vBoWPos)
IdxPos, DistNeg = findnn(histogram, vBoWNeg)
if (DistPos < DistNeg):
    sLabel = 1
else:
    sLabel = 0
return sLabel
```

1.3 Results

I selected $k = 10$ and $numiter = 10$ and on the first run on Colab I obtained 88% and 96% accuracies for positive and negatives images respectively. However, when running locally on the computer, these accuracies with the same parameters dropped to 71% and 64% (Fig. 1). As K-means chooses the centers randomly at the beginning, it can be due to chance that in the first case they reached the correct values by 10th iteration and in the second they did not.



((a)) Accuracies when run on Colab



((b)) Accuracies when run locally.

Figure 1: Bag-of-Words accuracy checks.

2 CNN-based Classifier

2.1 Implementation

2.1.1 A Simplified version of VGG Network

In accordance to the simplified VGG network description, the following architecture was implemented:

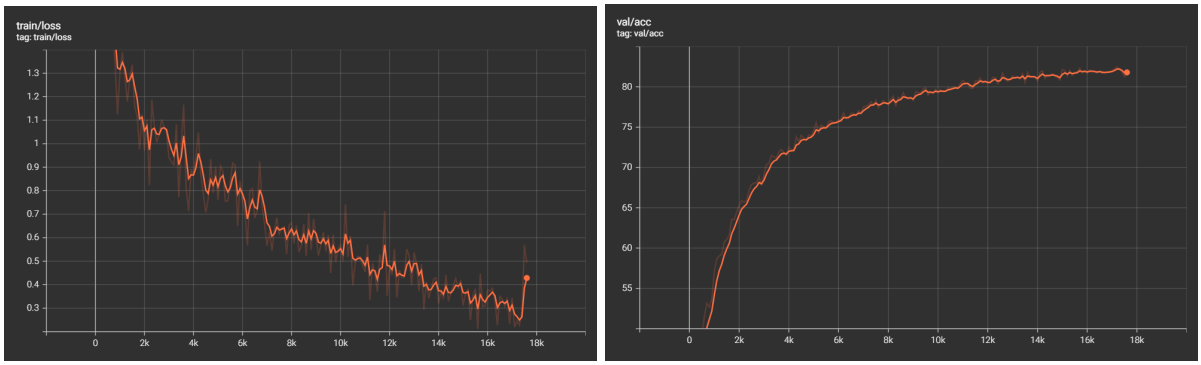
1. 2D Convolution Layer: 3 input channels, 64 output channels, kernel of size 3, stride equal to 1, padding equal to 1 (size change: 3x32x32 → 64x32x32).
2. 2D Max Pooling: kernel of size 2, stride equal to 2, padding equal to 0 (dimensions change: 64x32x32 → 64x16x16).
3. 2D Convolution Layer: 64 input channels, 128 output channels, kernel of size 3, stride equal to 1, padding equal to 1 (dimensions change: 64x16x16 → 128x16x16).
4. 2D Max Pooling: kernel of size 2, stride equal to 2, padding equal to 0 (dimensions change: 128x16x16 → 128x8x8).
5. 2D Convolution Layer: 128 input channels, 256 output channels, kernel of size 3, stride equal to 1, padding equal to 1 (dimensions change: 128x8x8 → 256x8x8).
6. 2D Max Pooling: kernel of size 2, stride equal to 2, padding equal to 0 (dimensions change: 256x8x8 → 256x4x4).
7. 2D Convolution Layer: 256 input channels, 512 output channels, kernel of size 3, stride equal to 1, padding equal to 1 (dimensions change: 256x4x4 → 512x4x4).
8. 2D Max Pooling: kernel of size 2, stride equal to 2, padding equal to 0 (dimensions change: 512x4x4 → 512x2x2).
9. 2D Convolution Layer: 512 input channels, 512 output channels, kernel of size 3, stride equal to 1, padding equal to 1 (dimensions change: 512x2x2 → 512x2x2).
10. 2D Max Pooling: kernel of size 2, stride equal to 2, padding equal to 0 (dimensions change: 512x2x2 → 512x1x1).
11. Linear layer: 512 input features, 128 output features.
12. ReLU function application.
13. Dropout with probability of 0.5.
14. Linear layer: 128 input features, 10 output features.

The code implementation of this part is presented below:

```
layers = []

layers.append(nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1)) # 3x32x32->64x32x32
layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0)) # 64x32x32->64x16x16

layers.append(nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)) # 64x16x16->128x16x16
layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0)) # 128x16x16->128x8x8

layers.append(nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)) # 128x8x8->256x8x8
layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0)) # 256x8x8->256x4x4

layers.append(nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1)) # 256x4x4->512x4x4
layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0)) # 512x4x4->512x2x2

layers.append(nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)) # 512x2x2->512x2x2
layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0)) # 512x2x2->512x1x1

layers.append(nn.Flatten())
layers.append(nn.Linear(in_features=fc_layer, out_features=128))
layers.append(nn.ReLU())
layers.append(nn.Dropout(p=0.5))
layers.append(nn.Linear(in_features=128, out_features=classes))

self.model = nn.Sequential(*layers)
```

2.2 Results

After running the training for 50 epochs, we can see the accuracy and loss changes in Fig. 2. The test result accuracy was equal to 81% (Fig. 3), which is an acceptable level of accuracy.



((a)) Loss during training ((b)) Accuracy value during training

Figure 2: Tensorboard screenshots.

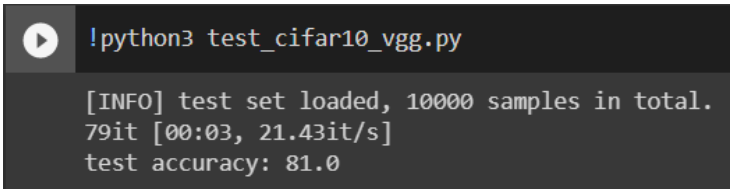


Figure 3: Test result.