Computer Vision Assignment 5

Nicolas Wicki

December 2021

1 Environment Setup

Everything worked as expected.

2 Bag-of-words Classifier

2.1 Local Feature Extraction

2.1.1 Feature Detection - Feature Points on a Grid

Based on the image width and height, I implemented a function returning equally spaced grid points placed on an image with a defined border (additional space outside the grid). [1]

2.1.2 Feature Description - Histogram of Oriented Gradients

Implementing the computation of the angles was not too clear to me, but after some reading up in the theory I could easily do so. The extraction of the cells according to the cell width and height was tedious, as was the extraction of the 4×4 blocks of cells. I really would have like to use `PyTorch` to accelerate the process using a GPU, but this part kind of made it impossible. Computing the histogram was drastically simplified once I encountered the pre-implemented functions by `numpy` and `PyTorch`.

2.2 Codebook Construction

The description of the codebook made it seem as though one would have to implement the whole process by oneself. Realizing, it was only the application of already implemented functions was a great relief. Finding fitting parameters for the amount of clusters and the maximum amount of iterations can be started using the default values of the `KMeans` function provided by `sci-kit learn`.

[1] Most of it did not pose too much of a problem. However, the confusion between x/y coordinates compared to image coordinates with row/column indexing did not happen just once.

2.3 Bag-of-words Vector Encoding

2.3.1 Bag-of-words Histogram

Tensor arithmetic and the `bincount` function provided by `numpy` and `PyTorch` made the implementation fast and efficient. However, finding the correct function is not always easy.

2.3.2 Processing a Directory with Training Examples

Here, we are simply asked to use the functions we just implemented. The only problem I encountered, was using the function `asarray()` provided by the `numpy` library. For some reason, it could not always interpret a list of `numpy` arrays as a 2D `numpy` array. To tackle this issue, I instead used `stack()` by `numpy`, which delivered consistent error-free results.

2.4 Nearest Neighbor Classification

Tensor arithmetic again made the implementation very efficient (in terms of runtime and code written). There is not much more to say here, and I will let the code speak for itself.

3 CNN-based Classifier

3.1 A Simplified Version of the VGG Network

I implemented the network using `Sequential()` as provided by `PyTorch` using some helper functions to abbreviate each layer collection (`Conv2d() + ReLU() + MaxPool2d()`). Sadly, the matching of the tensor sizes after the last convolution layer with the input size of the first linear layer poses some problem, as it cannot convert it from size $128 \times 512 \times 1 \times 1$ to 128×512 automatically. To still use the function `Sequential()` for the whole architecture, I implemented a `View` module to reinterpret the input to the linear layer. Adjusting the padding and stride of different layers to produce the desired image sizes can easily be done using the provided formulas in the `PyTorch` documentation.

3.2 Training and Testing

3.2.1 Training

For training, I used an RTX 3060 NVIDIA GPU, which decreased the runtime per iteration immensely when compared to CPU only. Also, using more `num_workers` improved runtime by a good amount. I trained the model for 300 epochs. The training loss is visible in Figure 1 and the validation accuracy in Figure 2. As can be seen in those figures, the measure converges after around 25 k iterations, which are around 75 epochs.

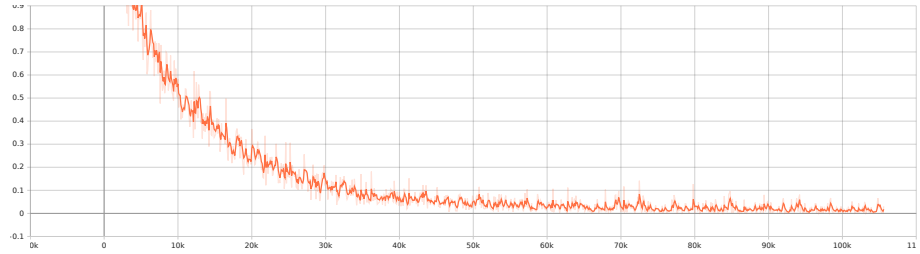

Figure 1: Training loss of the simplified VGG network

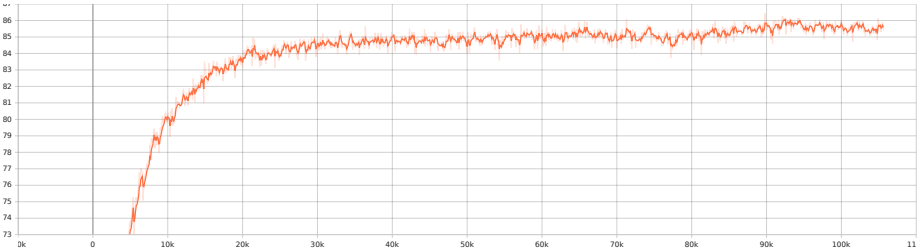

Figure 2: Validation accuracy of the simplified VGG network

3.2.2 Testing

The resulting test accuracy of the model is 84.73.