Assignment 4 – Computer Vision 1 – WS17/18 15/01/2018

Prof. Stefan Roth
Faraz Saeedan
Jochen Gast

This assignment is due on February 19th, 2018 at 13:00.

Common function module

For this assignment please make use of the functions given in `Common.jl`, which is imported at the top of each Julia problem file.

Packages

You may need to install the Julia packages `Clustering`, `MultivariateStats`, and `Optim`.

Group work and grading policy

You are required to work on each assignment in groups of ***two***. It is up to you to form groups, but please note that the group assignments cannot change after the first homework assignment. However, if too many students decide to work on their own, we may reassign groups manually for subsequent assignments.

Programming exercises

For the programming exercises you will be asked to hand in Julia code. Please use version **Julia v0.5** as we will use this version to test your solution. You need to comment your code in sufficient detail so that it will be easily clear to us what each part of your code does.

Your code should display your results so that we can judge if your code works from the results alone. Of course, we will still look at the code. You **must** adhere to the naming scheme for functions and files included with each problem. Do *not* rename function files and do *not* change the given function signatures. If you feel that there is a mistake in the assignments, ask us on Moodle.

Pen & paper exercises

There may be some theoretical exercises to do in the assignments. In this case we would greatly appreciate if you could typeset the theoretical part of your solution (ideally with LaTeX) and submit it as a PDF along with the rest of your solution. If you are not sufficiently familiar with a mathematical typesetting system such as LaTeX, you can also hand in a high resolution scan of a handwritten solution. Please write neatly and legibly.

Files you need

All the data you require to solve the given problems will be made available on Moodle `https://moodle.tu-darmstadt.de/course/view.php?id=11489`.

Handing in

Please upload your solutions in the corresponding section on Moodle. Each problem will specify which files should be included in your submission. You only need to submit one solution per group. If, for whatever reason, you cannot access Moodle get in touch with us as soon as possible.

Upload all your solution files (the writeup, `.jl` files, and other required files) as a single `.zip` or `.tar.gz` file. **Please note that we cannot accept file formats other than the ones specified!**

Late submissions

We will accept late submissions, but we will take *20% of the total reachable points* off for every day that you are late. Note that even 15 minutes too late will be counted as being one day late! After the exercise has been discussed in class, you can no longer hand in.

Other remarks

Your grade will depend on various factors. Of course, it will be determined by the correctness of your answer. But it will also depend on a clear presentation of your results and good writing style. It is your task to find a way to *explain clearly how* you solved the problems. Note that you will get grades for the solution, not for the result. If you get stuck, try to explain why and describe the problems you encountered – you can get partial credit even if you did not complete the task. So please hand in enough information for us to understand what you did, what things you tried, and how it worked! We will provide skeleton code for every assignment. Please use the provided interfaces as it allows us to better understand what you did.

We encourage interaction about class-related topics both within and outside of class. However, you should not share solutions with other groups, and **everything you hand in must be your own work**. You are also not allowed to plainly copy material from the web. You are required to **acknowledge any source of information that you used to solve the homework** (i.e. books other than the course books, papers, etc.). Acknowledgments will *not* affect your grade. Thus, there is no reason why you would not acknowledge sources properly. Not acknowledging a source that you have used, on the other hand, is a clear violation of academic ethics. Note that the university as well as the department is very serious about plagiarism. For more details please see `http://www.informatik.tu-darmstadt.de/index.php?id=202` and `http://plagiarism.org`.

Problem 1 - Optical Flow (8 points)

For estimating the optical flow the method of Lucas and Kanade minimizes the local energy

$$E\big(\mathbf{u}(x_0, y_0)\big) = \sum_{(x,y)\in\mathcal{N}_\rho(x_0,y_0)} \Big(f_x(x,y,t)u + f_y(x,y,t)v + f_t(x,y,t)\Big)^2,$$

where u and v are horizontal and vertical components of optical flow vector $\mathbf{u}$, $\mathcal{N}_\rho(x_0, y_0)$ is the spatial neighborhood window of size ρ around $(x_0, y_0)^{\mathrm{T}}$. By setting the derivatives w.r.t. u and v to zero, we can get following systems of equations:

$$\begin{bmatrix} \sum_{\mathcal{N}_\rho} f_x^2 & \sum_{\mathcal{N}_\rho} f_x f_y \\ \sum_{\mathcal{N}_\rho} f_x f_y & \sum_{\mathcal{N}_\rho} f_y^2 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum_{\mathcal{N}_\rho} f_x f_t \\ -\sum_{\mathcal{N}_\rho} f_y f_t \end{bmatrix}.$$

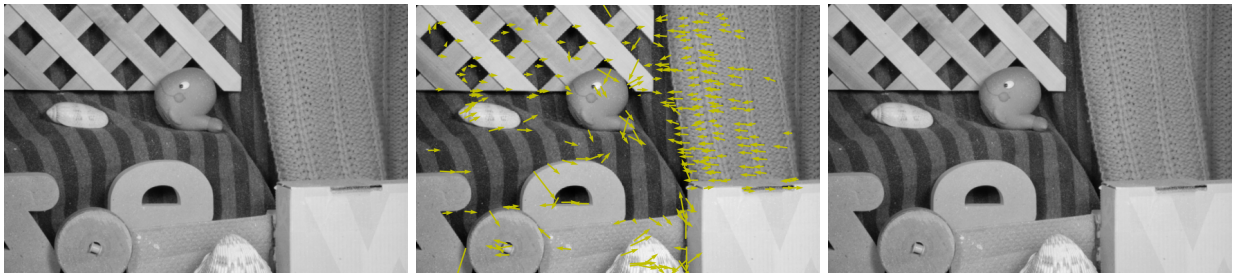
For simplicity, we can replace the hard window $\mathcal{N}_\rho$ by a convolution with a Gaussian smoothing filter G_ρ and get

$$\underbrace{\begin{bmatrix} G_\rho * (f_x^2) & G_\rho * (f_x f_y) \\ G_\rho * (f_x f_y) & G_\rho * (f_y^2) \end{bmatrix}}_{A} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -G_\rho * (f_x f_t) \\ -G_\rho * (f_y f_t) \end{bmatrix}.$$

Note the matrix A is just the structure tensor that we have used to compute Harris function values.

Tasks:

Compute the optical flow from the 9th frame (`frame09.png`) to the 10th frame (`frame10.png`) of the rubber-whale sequence and show the optical flow (at interest point locations) as illustrated below:



- As optical flow can only be reliably estimated at points, for which A has full rank we do not compute the optical flow for all image locations, but only at Harris interest points. The given skeleton already computes the interest point locations for you, so you do not have to worry about this.
- Implement `presmooth`, which smoothes images with a Gaussian filter, here we use the standard deviation $\sigma = 2.0$ and a filter size of 25×25.
- Implement `compute_derivatives` that computes first-order derivatives from the smoothed images. Use central differences for spatial derivatives and forward differences for the temporal derivative, respectively. All filtering operations should be done with "replicate" boundary conditions.
- Implement `compute_coefficients` that extracts the required coefficients of the linear system to solve for the unknown optical flow. As explained above, each coefficient should be smoothed beforehand, which essentially corresponds to the application of a window function; *i.e.* giving locations in closer proximity a higher weight. For the Gaussian window G_ρ, use a standard deviation of $\sigma = 2.0$ and a window size of 11×11. All filtering operations should be done with "replicate" boundary conditions.
- Implement `compute_flow` that solves the linear system of equations defined by the coefficients. The result should be optical flow components, both horizontal and vertical, for the given interest point locations.
- Finally, implement `show_flow` visualize the estimated optical flow vectors on top of the first image. Again, we only visualize optical flow at interest point locations. For visualization you can use show the image and subsequently call the `quiver` function of the `PyPlot` package. Do you get a similar visualization as above? If not, better check the signs of your computations :-) !

Part II - Classification

In this part you will implement a simple image classifier that categorizes images as either airplanes or motorbikes. To classifying images, we rely on two basic design choices:

1. Which *feature representation* do we use to encode our images?
2. Which *classification model* do we use to classify the encoded images?

Figure 1 below shows the process of classifying an image. In this assignment we will use the Bag-of-Words (BoW) representation for describing an image and a Multi Layer Perceptron (MLP) for the classification part.

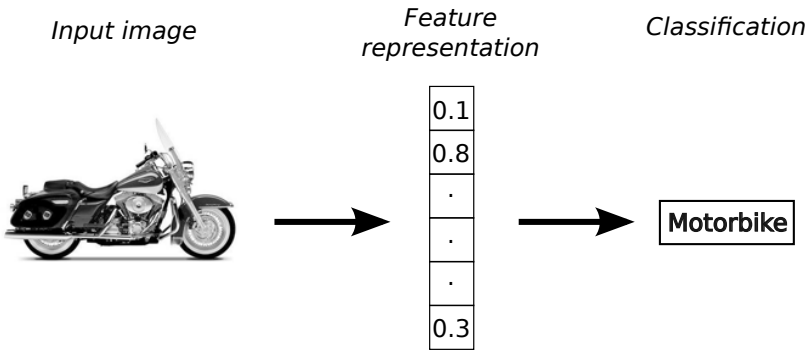


Figure 1: An image is first turned into a feature vector that is then subsequently classified.

Problem 2 - Bag-of-Words Model (15 points)

In this problem you will implement a simplified Bag-of-Words model. The main idea of the BoW representation is to first detect image features on the image and then use them to encode the image as a distribution over features of a codeword dictionary. Therefore, the image features are each assigned to the closest features of the dictionary and a histogram is used to represent the distribution. With this strategy, we can turn an arbitrary number of image features into a fixed-size histogram. This suits subsequent classification as most classifiers assume a fixed-size input.

Creating the codeword dictionary:

First, we create the codeword dictionary. It is a set of k key point descriptors and we obtain it by applying k-means clustering on *all* feature vectors from *all* the training images. K-means clustering aims to partition n observations $(\mathbf{x}_1, \cdots, \mathbf{x}_n)$ into k $(k \leq n)$ clusters $(s_1, \cdots, s_k)$ as to minimize the within-cluster sum of squares

$$\sum_{i=1}^{k} \sum_{\mathbf{x}_j \in s_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2, \tag{1}$$

where $\boldsymbol{\mu}_i$ is the mean of points in s_i.

Tasks:

- You are given image folders for both airplanes and motorbikes. Please implement `loadimages` that creates both a training and a testing structure containing the images, class labels, and number of samples per set. You should randomly separate the two sets of images (*i.e.* including airplanes or motorbikes, respectively) into two equal-size parts for training and testing. Please denote airplanes with class label $y = 0$ and motorbikes with class label $y = 1$.
- Apply the Harris detector and SIFT descriptor to obtain feature vectors for all images in both training and testing set. Use the parameters $\sigma = 1.4$, filter size 15×15 and threshold 10^{-7} for interest point detection and ignore points within a 10-pixel wide boundary. Please implement the feature computation for a single image in `im2feat` and the feature computation for the whole set in `extractfeatures`, which calls `im2feat` for all images.

- To get a database of image features, concatenate the features of all images in the training set by implementing the function `concatenatefeatures`. The resulting feature matrix will be used to compute the codebook of key points.
- Implement `computecodebook` that computes the codeword dictionary using k-means clustering with $k = 50$. The entries of the codebook are given as the centers of the k clusters. You can use the package "Clustering.jl" that already implements k-means.
- For all images you need to compute the histogram over assigned codewords. Therefore, implement the function `computehistogram` that takes a set of images as well as a codebook and computes a histogram for each image. Here, the histogram displays the number of occurrences of the codewords in an individual image. Use the (squared) Euclidean distance to assign feature vectors to the closest codeword in the codebook and normalize the histogram in the end such that it sums to one. Finally, the function `computehistogram` returns a so-called feature matrix in which each column contains the histogram of one of the images.
- Visualize the feature matrix in `visualizefeatures` by projecting the feature vectors onto the 2D plane. Use Principal Component Analysis for the projection (either use your implementation from assignment 2 or the Julia Package "MultivariateStats"). Plot the projected points and color the features belonging to bicycles in red and those belonging two airplanes in blue.

Problem 3 - A Multilayer Perceptron (25 + 5 points)

In this problem, we will use a multilayer perceptron (MLP) for binary classification:

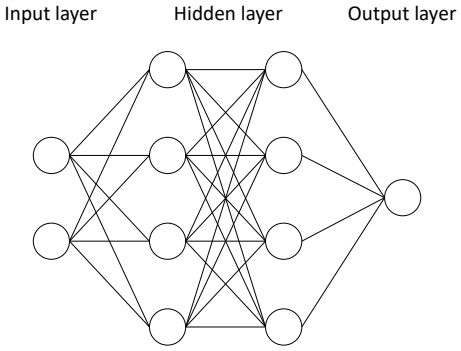


Figure 2: A simple MLP with 2 inputs, 2 hidden layers, and one output.

For a given training set consisting of N pairs of features and labels $(\mathbf{x}_i, y_i)$ with $y_i \in \{0, 1\}$, we can write the loss function of a MLP as

$$\begin{aligned} \text{logloss} &= -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right), \\ p_i &= \sigma_2(\mathbf{W}_2 \cdot \sigma_1(\mathbf{W}_1 \cdot \sigma_0(\mathbf{W}_0 \cdot \mathbf{x}_i + \mathbf{b}_0) + \mathbf{b}_1) + \mathbf{b}_2), \end{aligned} \tag{2}$$

where $\mathbf{W}_k$, and $\mathbf{b}_k$ are the weights and biases on the k-th layer respectively. These are the parameters that we want to learn from data. The inputs to every layer undergo a linear transformation and result is subsequently passed through a nonlinear activation function σ_k. Here, we want to use sigmoid activations on all layers, *i.e.*

$$\sigma_k(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}}}. \tag{3}$$

In the following programming assignments we will specify a network architecture by the number of nodes on each layer. For instance the net in Fig. 2 is specified `netdefinition` $= [2, 4, 4, 1]$. Please solve the following tasks.

Task 1: Training an MLP on toy data

On a 2D toy problem, we will first train an MLP with only 4 neurons on a single hidden layer.

- In the folder you will find two training data sets called `separable.jld` and `nonseparable.jld` containing 2D data points as well as their corresponding class labels. Implement the function `loaddata` that loads the data and the labels.
- To gather a little insight about the data structure, implement the function `showbefore` that shows the 2D data points as a scatter plot. Please color points with label 0 as blue and points with label 1 as red and include a legend that indicates points with corresponding class label.
- Implement the function `initWeight` that initializes all weights and biases of the MLP that structure is defined in the array `netdefinition`. The function should sample from $\mathcal{N}(0, \sigma_W)$ for the weights $\mathbf{W}$ and from $\mathcal{N}(0, \sigma_b)$ for the biases $\mathbf{b}$. The array `netdefinition` contains the number of nodes per each layer.
- Implement two helper functions: `weightsToTheta` that concatenates weights and biases into a variable `theta` and the function `thetaToWeights` that decomposes and reshapes weights and biases from the variable `theta` again.
- Implement the sigmoid activation function and its derivative: `sigmoid` and `dsigmoid_dz`.
- Implement the two functions `nnloss` and `nnlossgrad` that evaluate the loss (forward pass) and its gradient w.r.t. all $\mathbf{W_k}$ and $\mathbf{b_k}$ (backward pass). The gradients can be computed compactly by applying the chain rule for differentiation. Deriving the gradient is a common source of subtle errors. Therefore, we suggest that you

first write down the gradient on paper and evaluate it on a simple example. Then compare these numbers to the result of your implementation of the gradient.

- Now train an MLP from the training data by implementing the function `train`. After initializing the weights, you need to minimize the log loss function in w.r.t. the parameters. You can use the function `optimize` of the package "Optim.jl" that implements generic optimization algorithms. You need to pass the loss function as well as the gradient function to `optimize`. Using LBFGS as optimizer should deliver good results.
- Implement the function `predict` that takes a set of data points $\mathbf{x}$, weight vectors $\mathbf{W_k}$ and offsets $\mathbf{b_k}$ and calculates the network output $\mathbf{p}$ as well as a class label for each data point. If the output of a data point is greater than 0.5, you should assign the class 1 to the i^{th} data point, and class 0 otherwise.
- Write a function `showafter` that plots the data points colored according to their class labels. Additionally, this function should also plot the decision boundary of the trained network. The function `contour` from the PyPlot package and the function `meshgrid` from `Common.jl` might be useful.

Hints: To avoid redundant code you might find it helpful to introduce a separate function for computing the forward pass and keeping track of all necessary quantities like intermediate linear and non-linear activations.

Task 2: Categorize images with an MLP

Now we will train an MLP to classify images of airplanes and motorbikes. If you have not successfully solved `problem2`, do not worry! We have provided you with precomputed features for the training and test images as well as corresponding class labels. The data is provided in the files `imgstrain_gold.jld` and `imgstest_gold.jld`. However, note that these do not necessarily represent a valid solution to `problem2`.

Your task is to find a suitable MLP structure with the goal to optimize performance on the test set, *i.e.* achieving a low test error. Try i) different numbers of layers and ii) different numbers of perceptrons in each layer and keep the best value in your solution.

Bonus task (5 points): What happens if you want to train a deep network (>5 layers) for the previous classification task? What is the issue? What can you do about that? Please fix these problems in a new file `problem2_deep.jl` and write a brief answer to the aforementioned questions at the top of the file.