Computer Vision - VU 183.585

Exercise Part: Assignment II

Scene Recognition & Face Mask Detection

Computer Vision Lab
Institute of Visual Computing and
Human-Centered Technology

Marco Peer - mpeer@cvl.tuwien.ac.at
Aline Schaufelberger - e12142881@student.tuwien.ac.at

WS 2024/25

General

This document summarizes the tasks and technical requirements for assignment 2 of the lecture and exercises (VU) Computer Vision at TU Wien in the winter term 2023/2024. The setup and framework for assignment 2 will be the same as it was for assignment 1. The virtual environment `A2.txt` is provided and should be installed to complete the assignment.

Due: 22.01.2024, 23:59 (CET)
Total achievable points: 15
Additional achievable bonus points: 5

Therefore, for the complete exercise part, you can achieve 45 points.

Deliverables

The assignments must be handed in as a .zip file through **TUWEL**. Make sure all questions in the notebook files are answered and the code runs without errors. If requested, save your image results to your directory. Questions which are not sufficiently answered in the final submission or the missing of result images, will lead to the deduction of points. Read the instructions in the assignment description and code comments carefully and do not hesitate to ask questions.

HINT: Do not forget to put your group name at the top of the notebook as well as in all python files.

TASK3 - Scene Recognition using Bag of Words (7 Points)

The aim of this task is to recognize scenes, by classifying images into scene categories as visualised in Figure 1.

Figure 1: Image examples for the ten categories of the scene recognition database.

Background

For this task you will use SIFT features again, however this time for classifying images into scene categories like bedroom, kitchen, street etc. One method to achieve this is called **bag of visual words model**. It is a classic technique for image classification inspired by models used in natural language processing. An image is described by the distribution of its small local structures, where all local structures of an image are assigned to a visual "word" based on a predefined "vocabulary". The model ignores the word arrangement (spatial information in the image) and describes an image by the histogram of the frequency of visual words. The visual word vocabulary is established by clustering a large corpus of local features. Please see Section 14.4.1 in [Sze11] for more details on category recognition with quantized features. In addition, Section 14.3.2 discusses the creation of vocabulary.

Instructions

1 - Data import and Preparation

The data for this task consists of scene images of ten categories with 160 training and 40 test samples per category (cf. Figure 1). Import all images per category and save the respective class labels as integer values. In particular, save all images into an array and all class labels into a second array, where `labels[i]` is the label corresponding to the image saved in `images[i]`.

2 - Build Vocabulary and Clusters

In contrast to the previous assignment, you do not need to obtain SIFT features at detected keypoints but rather densely sample them on a regular grid in the image. In this part you will extract dense SIFT features from all training images and collect them for K-means clustering afterwards.

For feature extraction use the function `features.extract_dsift(..)`. Please note that you do not necessarily need to extract a SIFT feature for every pixel for vocabulary creation, e.g. around 100 features per image are enough to capture the approximately correct distribution of SIFT features for the given image data. Hence, create a list that contains a grid of `cv2.Keypoint` based on the step size parameter and optionally select a random subset of features per image (the function `random.sample(..)` might be helpful).

Bag Of Words After the SIFT features have been collected, we use the bag of words algorithm to classify our images: Firstly, you should apply K-means clustering to define the visual words $\mathbf{c}_i,\ i = 1, \cdots, N_c$, with N_c the number of K-means clusters that is also the size of your vocabulary. You can use `KMeans` from `sklearn.cluster` for KMeans clustering. For example, if $N_c = 50$, then the 128 dimensional SIFT feature space is partitioned into 50 regions. For any new SIFT feature you observe, you can figure out which cluster $\mathbf{c}_i$ it belongs to as long as you save the centroids of the original clusters. Those clusters form the visual word vocabulary. If clustering takes too long, you can restrict the random starts and number of iterations of the fitting process.

3 - Classification

The next step is to build a feature representation for every image in the training set that can be used for the classification of unseen images later on. An image is represented by a histogram of visual words, which means that all SIFT features of an image are assigned to the nearest visual

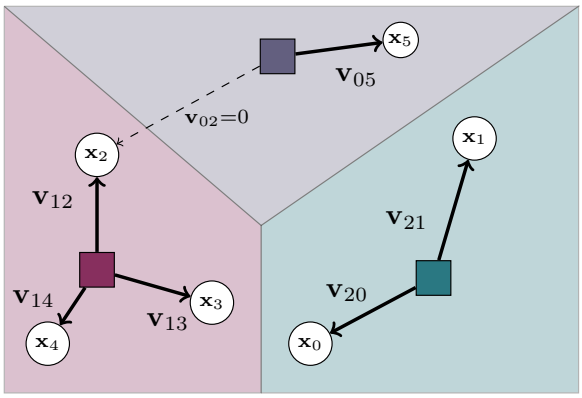

Figure 2: VLAD algorithm. Each local descriptor is assigned to its nearest centroid, and the sum of the residuals per centroid are calculated and concatenated for the final image descriptor [PKS23].

word, and the number of occurrences of every word is counted. In this step, the SIFT features should be more densely sampled than before. Thus, a good value for the step size is 2. Implement `count_visual_words(..)` in `features.py` in order to build a histogram of visual words. Then, build a kNN classifier where you fit your training data and later predict labels on the test set. Set the neighborhood size to 3.

4 - Evaluation

The last step in our pipeline is to classify all the images of the test set to investigate the classification power of the bag of visual words model for the classification task. Repeat the feature extraction and histogram preparation step with the test set, predict the test labels with the trained model of the previous step and plot the final result with `utils.plot_confusion_matrix(..)`. A score in the matrix at position (i, j) indicates how often an image with class label i is classified to the class with label j. Use the method `predict(..)` on your model to get the predicted labels and `score(..)` to get the overall accuracy. If you use the proposed hyperparameters, the bag of visual words method will achieve an overall accuracy of about $40\% - 50\%$ on the given dataset.

5 - Beyond Bag Of Words - Vector of Locally Aggregated Descriptors

Finally, we want to investigate an extension to the traditional bag of words model called *Vector of Locally Aggregated Descriptors* (VLAD), which not only considers the cluster assignment but also uses the distance to the nearest centroid (referred to as *residual*). It was originally proposed for image retrieval tasks by Arandjelovic and Zisserman in 2013 [AZ13]. In Figure 2, you see a schematic overview of the algorithm.

Algorithm The VLAD algorithm clusters a vocabulary to obtain N_c clusters $\{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{N_c-1}\}$ and encodes the local features of an image $\mathbf{x}_i$, $i \in \{0, \ldots, N-1\}$ via

$$\mathbf{v}_k = \sum_{i=0}^{N-1} \mathbf{v}_{k,i} = \sum_{i=0}^{N-1} \alpha_k(\mathbf{x}_i)(\mathbf{x}_i - \mathbf{c}_k), \quad k \in \{0, \ldots, N_c - 1\}, \tag{1}$$

with $\alpha_k = 1$ if $\mathbf{c}_k$ is the nearest cluster center to $\mathbf{x}_i$, otherwise 0. Therefore, for each descriptor $\mathbf{x}_i$, the algorithm calculates the residual $\mathbf{x}_i - \mathbf{c}_k$ to the nearest centroid. The final global descriptor $\mathbf{V}$ for the image is then obtained by summing up the residuals for each cluster separately and concatenating the vectors $\mathbf{v}_k$. The dimension of $\mathbf{V}$ is $N_c \cdot D$, with D the dimension of the local descriptors (for SIFT $D = 128$).

Implementation Implement the VLAD algorithm in `features.calculate_vlad_descriptors`. The final descriptor $\mathbf{V}$ for each image should be l_2-normalized via `sklearn.preprocessing.normalize`. Make sure to set up the training codebook with the VLAD descriptors of the training set and evaluate and compare the performance in the Jupyter notebook.

Deliverable

Submit all the files in a folder called `Task3_[GROUP_NUMBER]` including a folder: `results` with the image plots mentioned in the notebook. Put your group number on the image results and in the python files.

TASK4 - Face Mask Detection using Convolutional Neural Networks (8 Points)

In this task, you will deal with the basics of deep learning by applying a neural network to a specific problem, in our case **Face Mask Detection**. The task is to develop a neural network which takes an image and predicts whether the person in the image is wearing a face mask (binary classification). The dataset is publicly available (see `https://www.kaggle.com/omkargurav/face-mask-dataset`) and contains about 3750 images for each class. In Figure 3 a few images per class are shown. The train, validation and test split is already done for you.

Figure 3: Selected samples of the facemask dataset used.

Environment Setup

For this task, you will use **pytorch**[1], a library which offers high-level access to machine learning methods. Before you set up the virtual environment for this task, read the following setup instructions carefully: You can use your graphics card to train your model, which is recommended and will shorten the training time significantly. If you have a GPU that supports CUDA, check the hardware requirements[2] (you need a valid CUDA installation) and install the required software[3] according to your operating system. Use `A2.txt` to create your environment. If you do not have a GPU with CUDA support or no GPU is available at all, the task is still feasible. You can check if your GPU is ready to use in Python via

```
torch.cuda.is_available()
```

[1] `https://pytorch.org/`
[2] `https://developer.nvidia.com/cuda-zone`
[3] `https://pytorch.org/get-started/locally/`

Hardware Limitations

In case your hardware is not powerful enough, you have a few options for Task 4:

- Decrease input image size.
- Decrease the training dataset (e.g. only use 70% of the data, should decrease accuracy only by a small amount).
- Check `num_workers` of dataloaders.

We highly recommend you to attend the tutor sessions in case you have problems regarding your hardware setup or your python environment.

Theory

A Convolutional Neural Network (CNN) is a special type of **artificial neural network**. The basic idea has been developed more than 20 years ago [LeC+99]. Because of the technological boost in the recent years and the increased observable performance of CNNs in different applications, this technology got increased attention in the computer vision research community.
In general, the architecture of a CNN consists of one input layer, one output layer and several hidden layers in between, where the output of one layer is the input of the next one. The architecture of the hidden layers can be split in two consecutive parts according to their general functionality: (1) **feature learning** and (2) **classification**.

Summary of Basic CNN Components:

- **Input Layer:** Image $[C \times W \times H]$, where C is the number of image channels (e.g. for RGB images: $C = 3$)
- **Hidden Layers:**
  - **Layers for feature learning:** Convolutional and pooling layers for simple architectures. More advanced architectures include variations of convolutional layers e.g. Sparse CNN [Liu+15]
  - **Layers for classification purpose:** Depending on the architecture and the classification task, subsequent fully connected linear layers.
- **Output Layer:** class scores $[n \times 1]$

A CNN takes an image as input and provides an n-dimensional array as output, where n is the number of classes. The first proposed CNN architecture by Le Cun et al. [LeC+99] is visualised as an example in Figure 4: the convolutional layers are structurally different from the linear layers at the end of the network. The subsampling step is achieved through pooling layers, which reduce the dimension of the convolutional output. Read more in-depth information about how the layers work in Szeliski's computer vision book [Sze20] section *5.4 Convolutional neural networks*. A convolutional layer consists of several convolution kernels and outputs the input convoluted with the kernels, so called *activation maps* or *feature maps*. The aim is to learn the filter parametrisation in such a way so the network is able to extract the relevant features of the given dataset. The convolutional layer is usually followed by an activation function for non-linearity (e.g. Rectified Linear Unit (ReLu)).

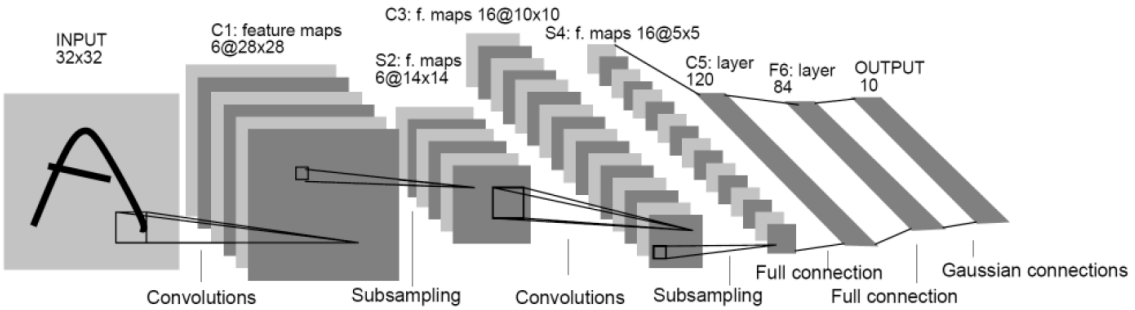

Figure 4: First proposal of a convolutional neural network LeNet-5.[LeC+99]

A CNN training process can be separated into two phases: the **forward pass** and **backward pass**. During the forward pass, the input image is passed and transformed through the layers. During the backward pass, also called **backpropagation**, the gradients are backpropagated, and weights of the layers are updated based on the used optimization strategy and loss function. The objective is to fit the network's trainable parameter to the training dataset with enough flexibility to generalize well over unseen data, while keeping the miss classification rate low. This problem is also called **Bias-Variance Tradeoff**.

Instructions

The network structure you are going to use for this task is already implemented in `my_model.py` and takes an input image with the dimensions [64 x 64 x 1] and outputs a number (1 - mask, 0 - no mask).

1 - Image Import and Preparation

This time you do not need to manually import the data, but work with provided dataset/dataloading classes. Use the provided classes `FaceMaskDataset` and `DataModule` and their methods to create a data generator for train, validation and test dataset. Pleae make yourself familiar with the classes. With the generator, you can define image processing functions which will be applied at runtime when the iterator imports the images. Image preprocessing for deep learning models is crucial to ensure numerical stability. For training, the images need to be normalized as well as resized. You should use the `torchvision.transforms` library by creating a list of transformations applied via `torchvision.transforms.Compose([ListOfTransformations])` where `ListOfTransformations` is a list of transformations provided by the package (e.g. `ToTensor()`, `Resize()`). Refer to the https://pytorch.org/vision/stable/transforms.html for further explainations.

2 - Simple Mask Classifier CNN

In this subtask, you will train a convolutional neural network, the class already has been designed and implemented in `my_model.py`. The network properties are summarized in Figure 5, which consists of two convolutional layers, each followed by one max-pooling layer + ReLU as activation function and subsequently, for classification, one linear layer fully connected to the flattened output of the convolutional stage. Take a look at the network architecture in Figure 5 - you are supposed

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MaskClassifier                           --                        --
├─Conv2d: 1-1                            [32, 32, 62, 62]          896
├─MaxPool2d: 1-2                         [32, 32, 31, 31]          --
├─Conv2d: 1-3                            [32, 32, 29, 29]          9,248
├─MaxPool2d: 1-4                         [32, 32, 14, 14]          --
├─Linear: 1-5                            [32, 1]                   6,273
==========================================================================================
Total params: 16,417
Trainable params: 16,417
Non-trainable params: 0
Total mult-adds (M): 359.30
==========================================================================================
Input size (MB): 1.57
Forward/backward pass size (MB): 38.38
Params size (MB): 0.07
Estimated Total Size (MB): 40.02
==========================================================================================
```

Figure 5: The network structure of the CNN used for facemask detection.

to initialize the given layers in the `init` method in `my_model.py`. Pay attention to the configuration of the convolutional layers (e.g. kernel size, stride, padding). In general, you can extend/change the network architecture, make sure you can argue about the output shapes and you achieve an accuracy above 90%. Do not forget to add a `softmax` or `sigmoid` function right before the output layer.

For training and predicting, `pytorch` uses the so-called `forward` method where you can code the forward pass (which describes how the input is propagated, the backward pass will be internally dealt with when using built-in functions from `pytorch`). Do NOT initialize any kind of layer which has trainable parameters inside the forward method.

The objective of the optimizer is to adapt the network parameters to reduce the loss, and the loss function measures how far the network output is from the ground truth label. For this task, you will use a common optimizer called **Adam** which is based on stochastic gradient descent. For the loss function, you will use **binary cross-entropy loss** (`BCELoss`). This loss will train the CNN to output a probability value p. If $p \geq 0.5$, the input image will be labelled as class 1 (mask), otherwise as class 0 (no mask).

Before you start the actual training, there are several **hyperparameters** you have to define. The **number of epochs** represents the number of times your network has seen the complete training set, the **learning rate** will determine the step size of the optimizer. In general, there is no optimal range, and it is critical for convergence to choose the learning rate based on the type of your data and the loss function appropriately. If the learning rate is too small, the loss might decrease not fast enough, and if the value is set to high, the optimizer might jump local minima in the loss landscape. The batch size refers to the number of samples used for one optimization step (and is usually a power of two, mainly limited by your CPU/GPU memory).

3 - Regularization

One problem with learning-based models is **overfitting**, which is a common effect of a model not generalizing well. **Regularization** is a technique to reduce the variance and avoid overfitting to the training data. In this step, you will add dropout layers to the simple scene CNN. These layers randomly remove connections between layers per epoch and therefore allow more training flexibility. See more on regularization and how it works in the original publication by Srivastava et al. [Sri+14] Add the dropout layers in `my_model.py`.

A second approach for regularization and well-defined gradients is called batch normalization. [IS15] It standardizes your intermediate results (zero mean and standard deviation one) while actively learning a linear transformation to improve the speed of training as well as avoiding overfitting. Refer to the paper if you are interested in the theoretical background (The concept is quite old regarding current deep learning methods, there are also existing newer developments based on normalization, e.g. group normalization). You can simply put a batch norm layer (`BatchNorm2d` - `https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html`) after each convolutional layer.

4 - Data Augmentation

There are two options to decrease overfitting and increase the test accuracy even further: A deeper network with millions of parameters more, or a larger training dataset. The first is not feasible within this course, and for the second argument, we simply do not have more data. Instead, you will implement a common method to artificially increase training samples: by **data augmentation**. Augmenting data allows the network to be more robust against small variations in unseen data.

Extend the list of image preprocessing functions. Use `torchvision` again to extend the previously implemented preprocessing list. You are free to use any kind of data augmentation. Evaluate the model trained with augmented data and check if you can improve your results.

Create a new model and before you start training use `utils.plot_activation_maps(..)` to plot the `activation maps` and `kernels` of the untrained model. The activation maps show the input convoluted with kernels of the first convolutional layer. The kernels have random values prior training, but after the training, the values will change. Train the third model with the augmented training dataset iterator and plot the activation maps again after.

5 - Evaluation

At last, evaluate the performance of your last model using the 2×2 confusion matrix as you did in TASK3. You can use `sklearn.metrics.confusion_matrix` as well as the `predict` function of the provided trainer. Since there are two classes in the given scene dataset, the values of the ground truth labels and predicted labels should be zero or one. If you are interested, you can additionally visualize some images which were falsely classified.

Deliverable

Submit all the files in a folder called `Task4_[GROUP_NUMBER]` including a folder: `results` with the plots mentioned in the notebook. Put your group number on the image results and in the python files.

Optional: BONUS - Scene Recognition via Deep Learning

This year's bonus will cover scene recognition via deep learning. You can earn 5 extra points. Therefore, the total amount of achievable points within the lecture is **105** (45 exam and 60 examination).

Goal: You are supposed to design a deep learning approach for scene recognition while improving the results compared to the Bag of Visual Words method of Task 3. The kNN classifier you implemented in Task 3 should achieve, depending on the hyperparameters you have chosen (e.g. vocabulary size, number of SIFT samples,...), almost 60% overall accuracy with the given test set. The goal of the bonus task is as well scene recognition. Instead of handcrafting features for a classifier, you will use CNNs like in Task 4, which learn the feature distribution of the dataset by optimizing its parameters over time. This task aims to give you a high-level overview of the application of deep learning, in particular with CNNs. You will use the same dataset as Task 3, train a scene classifier and use it to predict the labels (scene categories) of the test set.

Generally, you are completely unconstrained regarding your dataloading, network architecture and training strategy although it is highly recommended to reuse and adapt some implemented classes of Task 4. However, we expect you to implement a few advanced strategies in order to make yourself familiar with state-of-the-art deep learning strategies. You will get one point for a proper main implementation (neural network, trainer etc.), one point for each strategy implemented (see below) and one point if you can improve the performance compared to the visual bag of words method. Do some evaluation regarding performance and compare it to Task 3 (confusion matrices, accuracy). **We only grant points if your documentation is clear and we can comprehend your results. You are also encouraged to include a .pdf with your main documentation and results.**

Regularization

As already applied in Task 3, you are supposed to use some kind of regularization, e.g.

- Dropout and Batch Normalization
- Data augmentation
- Weight decay (an intuitive introduction can be found here).

Transfer Learning

A popular approach for image classification, in particular on small datasets, is called transfer learning. Its principle is applying a network trained on one specific task for solving another problem, e.g. using a network pretrained on a huge benchmark dataset like ImageNet. Pretrained networks are often frozen (their parameters will not change) during the training process and serve as a so-called *feature extractor* since they are trained on highly versatile data and can extract prominent features of the input. You only have to deal with designing an appropriate bottleneck layer. For example, freeze your pretrained model, replace the last fully connected layer of the model by a proper dense layer for the classification task. Since all other parameters are frozen, you only train the last dense layer. Pay attention that you might need to normalize your data in a different way (e.g. the ImageNet mean and standard deviation).

Concerning `pytorch`, pretrained networks can be found at `https://pytorch.org/vision/stable/models.html`. You can freeze parameters of a model via

```
for param in model.parameters():
    param.requires_grad = False
```

where `model` describes your pretrained network instance. You can also investigate if fine-tuning (adapt the parameters of your pre-trained network as well) helps increasing the performance. In the following, a popular network architecture called residual neural networks is briefly described. Feel free to make yourself familiar with other common architectures you can use for transfer learning.

Residual Net: In 2015, He et al. proposed a novel network architecture which learns so-called residual functions. [He+15] The main principle is illustrated in Figure 6. A residual network prevents vanishing gradients by adding the input to its previously calculated output (*skip connections*). This has shown to increase accuracy in particular for deep neural networks. If the forward pass is described by the function f, the output of the residual layer yields

$$y = f(x) + x, \tag{2}$$

where x is a given input. The so-called residual function learned by the network is then defined by $f(x) = y - x$.

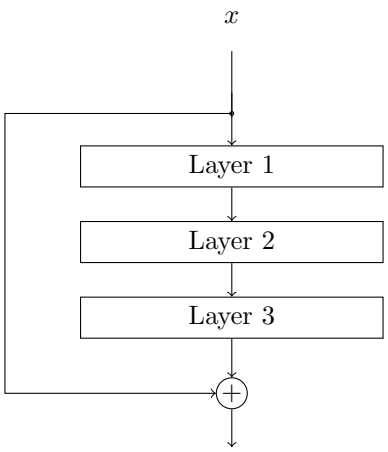

Figure 6: Example for a residual architecture with three layers

Learning Rate Scheduler

One of the most crucial hyperparameters for training is the learning rate. For example, stochastic gradient descent (SGD) is described by

$$w_{i+1} = w_i + \alpha \frac{\partial J(y, \hat{y})}{\partial w_i}, \tag{3}$$

where w_i is an arbitrary weight within the network, $J(y, \hat{y})$ the loss and α the learning rate. You can refer to the learning rate as a step size of the optimizer for updating the model's parameters.

Since in the beginning of the training the parameters are far away from the optimum, greater steps are required than in the end of the training process where the optimizer may only need to do small steps for converging to the optimum. Choosing a learning rate too big will cause the optimizer to be stuck while with a learning rate too small the optimization process will take longer.

Those problems can either be tackled by high-level optimizers such as Adam and/or by so-called learning rate schedulers - often both are combined. These schedulers are based on changing the learning rate depending on the current epoch or even the current step. Regarding `pytorch`, there are a variety of different optimizers existing as well as learning rate schedulers (see `https://pytorch.org/docs/stable/optim.html`). A popular approach for refining the weights is an exponential learning rate decay (see `LambdaLR`). Choose one and properly adapt the learning rate to improve the performance.

Hints

1. Start with implementing a dataloading suitable for multiclass classification. Considering the file structure of the dataset, we recommend using `torchvision.datasets.ImageFolder` (see `https://pytorch.org/vision/stable/datasets.html`) for loading your data (images as well as labels). Make sure you are coding as general as possible (e.g. regarding image size, a good starting point is 100×100, but you can try other sizes too). When using pretrained neural networks, familiarize yourself which kind of input data the network is expecting (e.g. shape and range of values) and adapt your preprocessing transformations.

2. When setting up your model, think about the output dimensions of your layers. You can also use `summary` from the `torchinfo` library for checking and debugging (refer to the error message telling you where possible mismatches are occurring).

3. Since you are dealing with multiclass classification, do not forget to choose a proper loss function (see `https://pytorch.org/docs/stable/nn.html#loss-functions`). Also calculate suitable metrics (e.g. accuracy) to evaluate the performance of your model.

Deliverables

You are unconstrained regarding the file structure. Please create a separate file for each implemented class (e.g. dataloading class, model class). If your main file is not a jupyter notebook, we expect you to submit some kind of documentation, e.g. a pdf document highlighting your contributions and results. Make sure your code runs with the provided virtual environment. Put your group number on any image result and in each file. Please also provide the saved **model with the best performance** so we are able to check your results if necessary. Upload your work as a `.zip` in TUWEL together with Task 3 and 4. Make sure to comment your code, any form of good coding (e.g. pep8 guideline) is appreciated.

References

[LeC+99] Yann LeCun et al. “Object Recognition with Gradient-Based Learning”. In: *Shape, Contour and Grouping in Computer Vision*. Berlin, Heidelberg: Springer-Verlag, 1999, p. 319. ISBN: 3540667229.

[Sze11] Richard Szeliski. *Computer vision algorithms and applications*. 2011. URL: http://dx.doi.org/10.1007/978-1-84882-935-0.

[AZ13] Relja Arandjelovic and Andrew Zisserman. "All About VLAD". In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2013. DOI: 10.1109/cvpr.2013.207. URL: http://dx.doi.org/10.1109/CVPR.2013.207.

[Sri+14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[He+15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[Liu+15] Baoyuan Liu et al. "Sparse Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

[Sze20] Richard Szeliski. *Computer Vision: Algorithms and Applications, 2nd ed.* 2020. URL: http://szeliski.org/Book/2ndEdition.htm.

[PKS23] Marco Peer, Florian Kleber, and Robert Sablatnig. "Towards Writer Retrieval for Historical Datasets". In: *Document Analysis and Recognition - ICDAR 2023*. Springer Nature Switzerland, 2023, pp. 411–427. ISBN: 9783031416767. DOI: 10.1007/978-3-031-41676-7_24. URL: http://dx.doi.org/10.1007/978-3-031-41676-7_24.