Technische Universität Berlin

Remote Sensing Image Analysis Group

Technical Project Documentation:

Content-Based Retrieval of Mars Images

David Hoffmann, Niklas Sprengel & Kim Schwarz

July, 2021

Contents

1 Setup **1**

1.1 Installation 1

2 Project Structure **2**

3 Usage **4**

3.1 MIRS 4

3.2 Baseline system 6

4 Components **7**

4.1 Dataset Classes 7

4.2 CBIRModel 7

4.3 Training 8

4.4 Testing 9

1 Setup

1.1 Installation

To install, clone the git repository at https://git.tu-berlin.de/rsim/cv4rs-2021-summer/CBIRforMars. The code was written for Python 3.9.5. Required libraries that need to be installed with pip can be seen in the following table:

library	version
torch	1.8.1+cu111
torchvision	0.9.1+cu111
tqdm	4.60.0
pytorch_metric_learning	0.9.99
numpy	1.20.2
Pillow	8.2.0
cupy-cuda111	9.0.0
matplotlib	3.4.1
sklearn	0.24.2

However, we expect that for most of these libraries the most current version should not pose problems. The only libraries where care might need to be taken are torch, torchvision and cupy-cuda. These need to be installed according to the cuda version your graphics card supports and they also have to match up to each other. Also for pytorch_metric_learning, we recommend using the exact version.
Lastly you need to copy the data set into the root folder. The data set is provided at https://tubcloud.tu-berlin.de/s/fYfckqqW2W2GwgP. You can also download the original DoMars16k data set at https://zenodo.org/record/4291940, we simply split up the `train` folder into a `train` and a `database` folder using a 50/50 split.

2 Project Structure

Our project `CBIR-Mars` ist structured as follows:

- `/basic-pipeline`: Contains the code for our baseline models.
  - `build_db.py`: Builds the image archive for image retrieval and stores it in `feature_db.json`.
  - `makeResultFile.py`: Make `result.json` containing the mAP score for the given model and the paths to the images retrieved for a number of given query images.
  - `query.py`: Takes the path to a query image as an input and retrieves the most similar images. However the `feature_db.json` file needs to be generated before running a query using this script.
  - `train.py`: Trains the baseline approach with triplet loss.
  - `train_classifier.py`: Trains the the baseline with cross entropy loss for classification.
- `/geomars-pytorch`: Contains the code for MIRS.
  - `/losses`: Implementations of the different Loss functions.
    - `hashing_loss.py`: Implementation of hashing loss.
    - `nt_xent.py`: Implementation of contrastive loss from https://github.com/Spijkervet/SimCLR.
  - `/models`: Trained model `.pth` files from the baseline setup to be used as feature extractors.
  - `/results`: Contains all result files utilised for our comparisons in the paper.
  - `CBIRModel.py`: Contains the model class for our network.
  - `build_db.py`: Builds the image archive for image retrieval stores it in `feature_db.p`.
  - `calcMAP.py`: Calculates the mAP@64 for a trained model. Requires the `feature_db.p` to be generated first.
  - `data.py`: Contains the Dataset implementations which load the multiview images and the images with their respective labels and also computes the k-means-clustering to return the grouping labels for the inter-class triplet generation.
  - `hparams.py`: Contains all important hyper-parameters and configuration parameters that can be adjusted for training the model and running queries.
  - `makeResultFile.py`: Makes `result.json` containing the mAP score for the given model and the paths to the images retrieved for a number of given query images. Calls `build_db.py` and `calcMAP.py` internally.

- `query.py`: Takes the path to a query image as an input and retrieves the most similar images from the hashcode database. Requires the `feature_db.p` to be generated first.
- `showResults.py`: Visualises `result.json`. For every query it stores the given image and the five most similar retrieved images in seperate `.jpg` files. This is used for evaluating the resulting rankings.
- `train.py`: Trains MIRS with the defined hyper-parameters.
- `utils.py`: Contains helper functions.
- `whitening.py`: Implementation of the adapted domain whitening layers heavily based on the implementation in https://github.com/roysubhankar/dwt-domain-adaptation.

3 Usage

3.1 MIRS

The code pertaining to the MIRS system is found in the folder `geomars-pytorch`. All scripts in this folder are designed to be executed from within this folder.
The modifiable hyperparameters and configuration parameters can be found in `hparams.py`. Following this section we provide a detailed overview of the function of each adjustable parameter in this script. The training and validation sets utilised in the training process are located in `data/train` and `data/val` respectively.

After setting the hyperparameters as desired, you should run train.py to train the hashing network. This will train the system based on the specified hyperparameters and plot the loss curves after finishing. The model with the best validation loss is then saved as `outputs/model_best.pth` and the model in the last epoch is saved as `outputs/model_last.pth`. By default all other scripts utilise `model_best.pth` by default and `outputs/model_last.pth` is only stored for debugging purposes.

Now that the model is trained, you should compute the image archive on which the image retrieval queries will be executed on. To do this, simply run `build_db.py`. This will load `models/model_best.pth` with the encoder specified in `hparams.py` and exectute it on every image in the `data/val` path. It then stores the generated archive containing the binary hash codes in a pickle file named `feature_db.p`.

After this you can now run `calcMAP.py` to calculate the mAP@64 for the trained model on the test set in `data/test`.
This will internally call the query function from `query.py` for every image in `data/test` and compute the average precision for the first 64 images in the returned rankings. These are then used to calcualte the mAP@64 for the entire test set. Alternatively you can run a specific query for a selected image by executing `python query.py PATH_TO_FILE`. This will visually show the 64 most similar images that were found in the archive in order of the computed ranking. It additionally outputs the average precision for this query as well as the average distance between the query and all output images in the command line. An example call would be

`python query.py data/test/cra/B02_010367_1631_XI_16S354W_CX2755_CY7546.jpg`.

To save the achieved metrics and some of the resulting rankings generated by a specific model for later evaluations we included a script which stores these in a `results/result.json` file. This script is named `makeResultFile.py` and internally calls `build_db.py`, `calcMAP.py` and `query.py` to compute overall metric results and the ranking results for some selected input images.

A table of all hyperparameters that can be specified in `hparams.py` and their effects as well as their value space is given in the following:

Name	Description	Value Space
LR	Learning rate	(0,1), but larger than 0.001 not advised
BATCH_SIZE	Batch size. Larger is generally better for this network, maximum is however limited by graphics card.	Positive integer
EPOCHS	Number of epochs to train for	Positive integer
DENSENET_NUM_FEATURES	Number of features in the last layer of feature extractor network. For DenseNet121 this is 1024.	Positive Integer
ADAM_BETAS	Beta values for Adam optimiser	Tuple of two floats in (0,1)
MARGIN	Margin for the triplet sampling process	(0,1)
HASH_BITS	Length of binary hashes outputted by the hashing network in bits.	Positive Integer
LAMBDA1	Weight for balancing loss for training the hashing network	Positive Number
LAMBDA2	Weight for push loss for training the hashing network	Positive Number
DENSENET_TYPE	What densenet to use as feature extractor. Implemented are DenseNet121s trained either on ImageNet, the DoMars16k data set for classification purposes or on the DoMars16k data set using triplet loss. The corresponding model files are either downloaded at execution or can be found under`models/`.	{"imagenet", "domars16k_classifier", "domars16k_triplet"}
INTERCLASSTRIPLETS	Toggles use of Interclass triplets in the training process. When MULTIVIEWS is set to `True` this has no effect.	Boolean
KMEANS_CLUSTERS	How many clusters KMeans calculates on the extracted feature vectors for the purpose of sampling inter-class triplets	Positive Integer
DOMAIN_ADAPTION	Toggles use of domain whitening layer between feature extractor and hashing network	Boolean
DA_GROUP_SIZE	Sets the group size for which the DWT layer will computed the covariance matrix.	Positive Number
MULTIVIEWS	Toggles between supervised and unsupervised approach. If `True`, unsupervised learning is used.	Boolean
PROJ_DIM	Dimension of projector head used in the unsupervised approach	Positive Integer
TEMPERATURE	Temperature parameter used in unsupervised approach	(0,1)

3.2 Baseline system

The baseline system can be found in `basic-pipeline`. It serves to compare the MIRS hashing models performance against a bare feature comparison model and to train feature extraction DenseNets for the MIRS model.

Two different types of feature extractors can be trained here. The first is trained by running `train.py` and utilises triplet loss to encode the disricimnative information in the outputted feature vectors. The second is trained by running `train_classifier.py` and trains the whole network as a classifier for the DoMars16k data set. The last fully-connected layer is then removed to utilise this network as a feature extractor for the baseline or the MIRS model. Both utilise a training and validation set located at `/data/train` and `/data/val` respectively. Similar to the MIRS model this model will also train for a set number of epochs and after each epoch the model is validated and the overall best model is then stored in `outputs/model_best.pth`. After training is finished the final model is stored in `outputs/model_last.pth`.

After the training is completed the resulting model can be used to create a feature vector archive by running `build_db.py`. By default `build_db.py` loads `outputs/model_best.pth` for the feature extraction. The feature extractor is applied to every image in `/data/database` and the encoded features are stored in `feature_db.json` with their respective sample names as keys.

Finally, you can run `calcMAP.py` to calculate mAP@64 or retrieve images from the archive similar to a query image by running `python query.py PATH_TO_FILE`.
Both of these work analogous to their counterparts described in Section 3.1 however to specify which kind of model should currently be used as the feature extractor the specific variables need to be modified within the code files as there is no central configuration file for the baseline system.

4 Components

4.1 Dataset Classes

The file `data.py` contains our custom PyTorch Dataset classes:

1. `MultiviewDataset`: Multi-view dataset for the unsupervised approach.
2. `ImageFolderWithLabel`: Data set to load paths for current images and theri corresponding labels. Also computes and provides the inter-class triplets if activated.

MultiviewDataset

A `MultiviewDataset` is initialised with the sample images and a set of transforms in `__init__`.

The `__getitem__` method allows to read a sample from the dataset. Every sample image is resized and normalised. `__getitem__` then returns two images. The first one is the resized and normalised original sample image, while the second one is an augmented view of the sample image. The augmented view is obtained by randomly applying augmentations from the set of transforms. We apply the same resizing and normalisation to the augmented view as to the sample image.

ImageFolderWithLabel

This Dataset class is responsible for loading the default images and their respective labels as well as the groupings when `INTERCLASSTRIPLETS` in `hparams.py` is set to `True`.

Its `__init__` method takes four input parameters. These are `interclasstriplets` which activates the the clustering for inter-class triplets, `n_clusters` which specifies the number of clusters for the k-means-algorithm and a `features_dict` which contains the pre-computed feature outputs from the feature extractor network. When `interclasstriplets` is set to `False` the `__getitem__` only returns the path of a random sample and the corresponding label. However when it is set to `True` it will first calculate a standardisation of all feature vectors given in `features_dict` and then compute the k-means-algorithm on these standardised features. The computed clustering is stored in the variable `self.kmeans_labels` and `__getitem__` then also returns the corresponding group index on every call.

4.2 CBIRModel

The content-based image retrieval (CBIR) model that represents our MIRS system is defined in `CBIRModel.py` in the class `CBIRModel`.

Its `__init__` function takes two boolean parameters, `useEncoder` and `useProjector`. If these are true, the DenseNet and respectively the projector head for the unsupervised approached are defined as part of the model. The reason `useEncoder` can be set to `False` is because in training, we calculate the DenseNet output for every training and validation image at the start for better performance. In the `train` method the network is then used on the stored DenseNet outputs and should therefore not use the DenseNet again.
Since `CBIRModel` is a superclass of `torch.nn.module`, the three fully connected layers and the LeakyReLU layers inbetween as well as the encoder and projector head are defined in `__init__`. The application of the model is then defined in `forward`, where all layers are sequentially applied and then finally the sigmoid function is used on the output.

4.3 Training

The training process is coded in `train.py`. This file contains two main functions, `train` and `validate`, and the code that calls these functions when the file is executed. This then initialises the model and all required data structures and calls the training and validation function appropriately.

The `train` function takes four parameters. `model` is the model that should be trained, in our case an instance of `CBIRModel`.
`dataloader` is the data loader that loads the images in batches. This data loader can be either an instance of `ImageFolderWithLabel` to load samples for supervised learning or an instance of `MultiviewDataset` for unsupervised learning.
Finally `train_dict` is the dictionary that contains the image paths of all training images as keys and the respective DenseNet outputs as values.
`train_dict_view2` is only used for unsupervised learning, where we use generated multiviews for the training images. Their paths are stored in this dictionary along with their respective DenseNet outputs.
We then iterate over the batches returned by the data loader. These consist of paths to the images in this batch and are used to extract the corresponding DenseNet outputs from `train_dict` and in the case of unsupervised learning also the DenseNet outputs for the multi-views of these images.
These feature vectors are then normally directly passed as inputs to the hashing model. However if `DOMAIN_ADAPTION` is set to `True` they are first passed through a domain whitening layer beforehand.

After this the following steps depends on whether we are doing supervised or unsupervised learning:

Supervised Learning: Next the outputs are L_2-normalized. We then build the triplets for our batch using the pytorch-metric-learning libraries online triplet sampling methods. If `INTERCLASSTRIPLETS` is `True`, we additionally generate a number of the inter-class labels based on the grouping labels provided by the `ImageFolderWithLabel` data set. Then the triplet loss is calculated on all of these triplets based on the corresponding labels. To this loss the hash losses which consist of the balancing and

push loss are are then added.

Unsupervised Learning: Every original image and the augmented view are inputted into the model. The outputs of the projection head are L_2-normalised and passed to the contrastive loss function to maximise agreement between the correlated image pairs. Also, the hash losses are calculated on the L_2-normalised output of the hashnet for the original image. The total loss in this case is then the sum of the contrastive and the hash losses.

If `DOMAIN_ADAPTION` is `True`, one more step is added to the above process. An additional `target_list` is generated before training, containing the feature outputs of the feature extractor network for each image in the target data set. A selection of features from this target data set is then passed through an additional domain whitening layer specific to the target data set. The outputs of this DWT layer are then passed to a so called `EntropyLoss` which evaluates their quality and is added to the overall loss function to update the internal weights of this target DWT layer accordingly. The final loss value is then backpropagated through the model using the `backward` function provided by `pytorch` and the parameters are updated using `optimizer.step()`.
The average loss over the whole training data set is also calculated and returned to later compute the loss curves.

`validate` works in the exact same way as `train` with the only differences being that gradient flow throughout the network is turned off for validation, that the loss is not backpropagated at the end and that the data set being used is the validation set instead of the training set.

4.4 Testing

To test our approach we implemented a pipeline of methods to build the hash code archive and run queries and tests on it. This is done in the files `build_db.py`, `query.py` and `calcMAP.py`.

The building of the archive is performed by the function `build_db()` from the file `build_db.py` which loads the model from the default path with the encoder specified in `hparams.py` and applies it to every image in the path `data/database/`. The resulting hash codes are stored in `feature_db.p` which can then be accessed by the `query()` defined in `query.py`. This function takes a path to a query image and and then calculates the hash code for this image using the same model with which the hashing archive has been build. By comparing the query hash code to all hash codes in then calculates the hamming distance to each entry and generates a ranking by sorting the found entries based on their distance. It then computes the average precision for the 64 first rankings and presents both metric and visual results to the user.

To compute the mean Average Precision over the whole test set of images the function `calc_map()` defined in `calcMAP.py` can be used. It computes the output rank-

ings for each image in the test set and calculates the mAP for all such returned rankings and presents its results on the console.

When all of these results should be stored for later analysis the script `makeResultFile.py` can be executed to save the computed mAP score as well as some example ranking results in a file called `results/result.json`. It internally utilises the above mentioned functions. Results stored like this can also be visualised with the script `showResults.py` which offers some rudimentary visualisation utilising the operating systems default image software.