Step 1: **Remove patients that look bad**

There were some patients that were misclassified, or their images were not clear on whether they had a fatpad tissue. To help the model perform better ([garbage in, garbage out principle](https://en.wikipedia.org/wiki/Garbage_in,_garbage_out)), we were provided with a list of patient IDs to remove. We can remove those patients using the following command (script):

**python3 remove\_bad.py --path\_dicom “PATH\_TO\_DICOM” --path\_to\_remove “PATH\_TO\_LIST” –path\_new *[optional]* “PATH\_TO\_MOVE”**

The script takes the following arguments:

**path\_dicom**:path to folder containing patient folders, which contain dicom images; folder structure:

>path\_dicom\_folder

>patient\_ID

image.dcm

[other images]

>[other patients]

**path\_to\_remove**: .txt file containing the patient IDs to remove, each line ending with a new line (including the last patient)

**path\_new**: *[optional]* path to place the removed patients. If not specified, the script will simply remove those patients from the dataset. If specified, patients will move the patients to the new folder.

Step 2: **Converting DICOM to PNG**

DICOM files are converted using the following script:

dicom\_converter.py

Running the script:

**python3 dicom\_converter.py --path\_dicom "PATH\_TO\_DICOM" --path\_excel "PATH\_TO\_EXCEL" --path\_output "OUTPUT\_FOLDER"**

The script takes the following arguments:

**path\_dicom**: path to folder containing patient folders, which contain dicom images; folder structure:

>path\_dicom\_folder

>patient\_ID

image.dcm

[other images]

>[other patients]

**path\_excel**: path to excel file containing information about dataset

**path\_output**: path where to place .png images; will split data into *negative*/*fracture*/*fatpad*/*both* folders.

The script automatically places the images into the following classes, using the excel file:

**negative**: fatpad negative, fracture negative

**fracture**: fatpad negative, fracture positive

**fatpad**: fatpad positive, fracture negative

**both**: fatpad positive, fracture positive

Step 3: **Split images into train, validation and test sets.**

For the Fatpad project, a split of 60%-20%-20% ratio for train-valid-test sets was used. For binary classifications, the following classes were used:

positive class: fatpad and both

negative class: fracture and negative

Running the script:

**python3 file\_mover.py --path\_dataset "PATH\_PNG\_DATASET" --path\_to\_split "DESTINATION\_SPLITS" --train\_ratio [0-100] --valid\_ratio [0-100] --test\_ratio [0-100]**

The script takes the following arguments:

**path\_dataset**: path to folder containing both/fatpad/fracture/negative folders, which contain patient folders; which contain png images; folder structure:

>path\_dataset

>both

>patient\_ID

image.png

[other images]

>[other patients]

>[other classes]

**path\_to\_split**: path where to place the splits; will create train/valid/test folders, and output .png images directly (no more separate patient folders). Each image will contain a prefix to mark the original, the patient’s ID, and an image ID.

**path\_output**: path where to place .png images; will split data into *negative*/*fracture*/*fatpad*/*both* folders.

**train\_ratio, valid\_ratio, test\_ratio**: integer values between 0 and 100, representing percentage of patients in each split. Their mustn’t be above 100.

*Note: If necessary, this script could be modified to save which patient ID is moved to which split.*

Step 4: **Build VGG16 model, train with frozen base**

From the VGG16 model with ImageNet weights, the first 4 convolutional blocks (base) are added to a new sequential model. Then 4 linear layers and an output layer are added to this. The base blocks are frozen, and the linear layers are trained for 256 epochs using a learning rate of 1e-05.

This script uses the adam optimizer with binary crossentropy as loss. This script also monitors the AUC score.

The script uses the checkpoint callback with save\_best\_only i.e. the script only saves the model with the highest AUC score.

At the end, the best checkpoint and the losses of the whole training is saved.

Running the script:

**python3 train\_base.py --model\_name "NAME" --epochs [int] --learning\_rate [float] --batch\_size [int] --path\_dataset "PATH\_TO\_DATA" --path\_model "PATH\_SAVE\_MODEL"**

The script takes the following arguments:

**model\_name**: Used to differentiate between models.

**epochs**: How long to train the model for (default: 256 epochs)

**learning\_rate**: Learning rate of the optimizer (default: 1e-05)

**batch\_size**: batch size (default: 32)

**path\_dataset**: path to dataset with train/valid folders with positive/negative classes

**path\_model**: path where to save the models (the model checkpoint and losses will be saved using path\_model and model\_name)

Step 5: **Unfreeze all layers, fine-tune the model further**

From the trained VGG16 model with frozen base, the base is unfrozen and the model is further trained for 128 epochs, using the checkpoint and early stopping callback.

The checkpoint callback monitors for the highest AUC score.

The early stopping callback monitors the validation binary crossentropy loss, and has a patience of 20.

Training stops when either the loss doesn’t improve for 20 epochs (via early stopping) or after 128 epochs of training.

Running the script:

**python3 finetune.py –prev\_model \_name “PREV\_NAME” --model\_name "NAME" --epochs [int] --learning\_rate [float] --batch\_size [int] --path\_dataset "PATH\_TO\_DATA" --path\_model "PATH\_SAVE\_MODEL"**

The script takes the following arguments:

**prev\_model\_name**: the name of the model to load and finetune

**model\_name**: Used to differentiate between models.

**epochs**: How long to train the model for (default: 128 epochs)

**learning\_rate**: Learning rate of the optimizer (default: 1e-05)

**batch\_size**: batch size (default: 32)

**path\_dataset**: path to dataset with train/valid folders with positive/negative classes

**path\_model**: path where to save the models (the model checkpoint and losses will be saved using path\_model and model\_name)

Step 6.1: **Test model performance**

This script loads a model, generates its predictions on the test data and then computes the confusion matrix and various metrics. These metrics are saved in *path\_model* with the other models, under file ***model\_name****\_metrics.csv*

Running the script:

**python3 compute\_checkpoint\_metrics.py --model\_name "NAME" --path\_model "PATH\_MODELS" ---path\_dataset "PATH\_TO\_TEST\_DATA"**

The script takes the following arguments:

**model\_name**: name of model to load

**path\_model**: folder which contains model checkpoints

**path\_dataset**: path to test data

Step 6.2: **Visualize model using GRAD-CAM**

This script outputs the heatmaps of the network using GRAD-CAM. A heatmap shows the areas of interest to the model i.e.which areas influenced the model’s prediction. For further information on GRAD-CAM, check the following link:

<https://keras.io/examples/vision/grad_cam/>

Running the script:

**python3 grad\_cam.py --model\_name "NAME" --path\_model "PATH\_MODELS" --path\_dataset "PATH\_TO\_TEST\_DATA" --path\_output "PATH\_TO\_PLACE\_HEATMAPS"**