Second Annual Data Science Bowl model documentation

Team Name: kunsthart
Names: Jeroen Burms, Jonas Degrave, Joni Dambre, Iryna Korshunova
Location: Gent, Belgium
Emails: jeroen.burms@ugent.be, Jonas.Degrave@UGent.be, irene.korshunova@gmail.com, joni.dambre@ugent.be
Competition: Second Annual Data Science Bowl

1. Background

Jeroen Burms, Jonas Degrave and Iryna Korshunova are PhD students at the Ghent University, iMinds Data Science Lab, department of electronics and information systems (ELIS), supervised by prof. Joni Dambre.
Jeroen, Jonas and Iryna participated in the National Data Science Bowl last year as a part of the winning team. Joni is new to Kaggle, but has used the platform to organise in-class competitions for her Machine Learning course.
We decided to enter the competition because we are all very interested in deep learning. In addition, we have experienced last year that Kaggle competitions are an excellent opportunity to try out the latest state-of the-art techniques and to exchange and transfer knowledge within our lab.
Jonas spent 2.5 months working on the competition, Iryna and Jeroen both spent about 1.5 months, Joni spent a few weeks.
We decided to team up because we are from the same lab, which makes competing separately almost impossible. Furthermore, we enjoy collaborating with each other and exchanging ideas.
Jonas set up the framework, and mainly worked on 2Ch and 4Ch models, slice projections scripts and ensembling scripts. Jeroen was mainly developing convnet models. Iryna developed another framework and worked on convnet models. Joni worked on the region of interest (ROI) detection.

2. Summary

Our solution is based on convolutional neural networks. We used fairly large convnets (up to 30 million parameters) with intensive use of data augmentation, dropout, and special network architectures to aggregate information from different sources. We used traditional computer vision techniques to identify the region of interest (ROI) in the images.

Our solution mainly differs from the other competitors in that we did not use hand-labelling or the Sunnybrook dataset. We only made use of the provided labels, and trained our models end-to-end (disregarding the ROI detection) using those.

Our winning final submission was an ensemble consisting of over 40 networks, which required around 3 days of training on 14 GPUs.

3. Models

3.1. Pre-processing, ROI detection and data augmentation

Our preprocessing pipeline made the images ready to be fed to a convolutional network by going through the following steps:

1) applying a zoom factor such that all images have the same resolution in pixel per millimeter
2) finding the ROI and extracting a patch centered around it
3) data augmentation
4) contrast normalization on each slice separately

We made use of the *PixelSpacing* metadata field to rescale the images to their physical size in millimeters. Afterwards, an image patch of a fixed size centered around the center of ROI was extracted and rescaled back to an image of 64 by 64 pixels for most of the models, sometimes we used an ellipsoidal ROI as a mask (see Figure 1). These patches were subject to random augmentation and contrast normalization and afterwards were used as convnets' inputs.

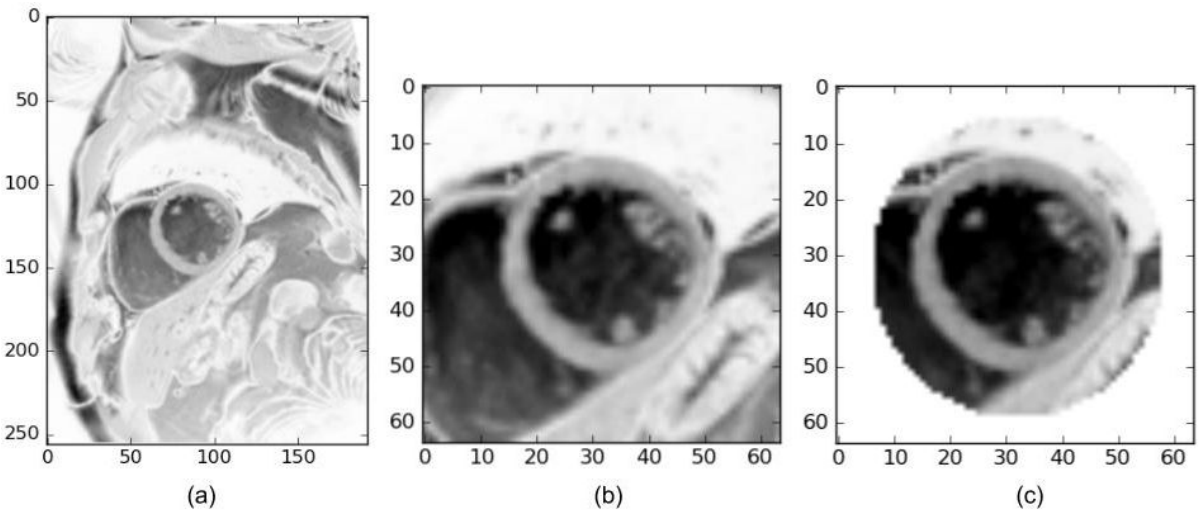
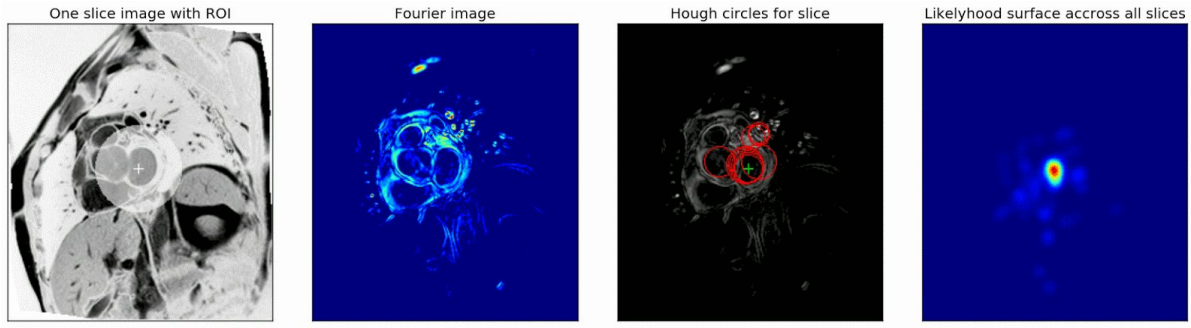


Figure 1. (a): sax_30 of patient 501 (b): extracted and rescaled patch (c): patch masked with a circular ROI

3.1.1. ROI detection

To locate the ROI in SAX slices we used classical computer vision techniques. For each patient, the center and width of the ROI were determined by combining the information of all the SAX slices provided. Figure 2a shows an example of the result (patient 1).



(a) (b) (c) (d)

Figure 2. ROI extraction steps

First, as was suggested in the Fourier based tutorial, we exploit the fact that each slice sequence captures one heartbeat and use Fourier analysis to extract an image that captures the maximal activity at the corresponding heartbeat frequency (Fig. 2b).

From these Fourier images, we then extracted the center of the left ventricle by combining the Hough circle transform with a custom kernel-based majority voting approach across all SAX slices. First, for each fourier image (resulting from a single sax slice), the *N* highest scoring Hough circles for a range of radii were found, and from all of those, the *M* highest scoring ones were retained. *N*, *M* and the range of radii are hyperparameters that severely affect the robustness of the ROI detection and were optimised manually. Fig. 2c shows an example of the *M* best circles for one slice.

Finally, a 'likelihood surface' (Fig. 2d) was obtained by combining the centers and scores of the selected circles for all slices. Each circle center was used as the center for a Gaussian kernel, which was scaled with the circle score, and all these kernels were added. The maximum across this surface was selected as the center of the ROI. The width and height of the bounding box of all circles with centers within a maximal distance (another hyperparameter) of the ROI center were used as bounds for the ROI or to create an ellipsoidal mask as shown in the figure.

Given these ROIs in the SAX slices, we were able to find the ROIs in the 2Ch and 4Ch slices by projecting the SAX ROI centers onto the 2Ch and 4Ch planes.

3.1.2. Data augmentation

We augmented the data to artificially increase the size of the dataset. We used various affine transforms. Some special precautions were needed, since we had to preserve the surface area. In terms of affine transformations, this means that only skewing, rotation and

translation was allowed. We also added zooming, but we had to correct our volume labels when doing so. This helped to make the distribution of labels more diverse, yet removes the a priori information of how likely a certain volume is. Next to these spatial transformations, we also included a random shift in the time domain.
Our best single models used the following augmentation parameters:

- rotation: random with angle between 0 and 360 degrees (uniform)
- translation x-axis: random with shift between -8 and 8 mm (uniform)
- translation y-axis: random with shift between -8 and 8 mm (uniform)
- rescaling: random with zoom factor between 67% and 150% (uniform)
- flipping: yes or no (bernoulli)
- time sequence shift: random with shift between 0 to 30 frames (uniform)

We augmented the data on-demand during training (real time augmentation), which allowed us to combine the image rescaling and augmentation into a single affine transform. The augmentation was all done on the CPU while the GPU was training on the previous chunk of data.

3.2. Network architecture

Most of our convnet architectures were strongly inspired by OxfordNet: they consist of lots of convolutional layers with 3x3 filters. We used 'same' convolutions (i.e. the output feature maps are the same size as the input feature maps) and non-overlapping pooling with window size 2.

We trained different models which can deal with different kinds of patients. There are roughly four different main kinds of models we trained: single slice models, patient models, 2Ch models and 4Ch models.

3.2.1. Single slice models

Single slice models are models that take a single SAX slice as an input, and try to predict the systolic and diastolic volumes directly from it. The 30 frames were fed to the network as 30 different input channels. The systolic and diastolic networks shared the convolutional layers, but the dense layers were separated. The output of the network could be either a 600-way softmax (followed by a cumulative sum), or the mean and standard deviation of a Gaussian (followed by a layer computing the cdf of the Gaussian). The general architecture of slice models is shown on Figure 3.

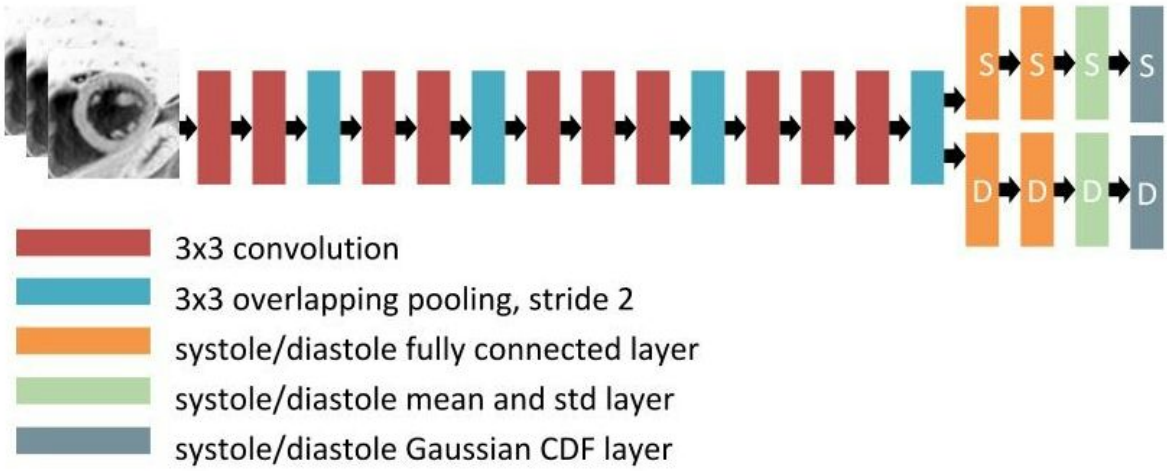


Figure 3. An example of a single slice model architecture

Although these models obviously have too little information to make a decent volume estimation, they benefitted hugely from test-time augmentation (TTA). During TTA, the model gets slices with different augmentations, and the outputs are averaged across augmentations and slices for each patient. Although this way of aggregating over SAX slices is suboptimal, it proved to be very robust to the relative positioning of the SAX slices, and is as such applicable to all patients.

Our single best single slice model achieved a local validation score of **0.0157** (after TTA), which was a reliable estimate for the public leaderboard score for these models.

3.2.2. 2Ch and 4Ch models

The 2Ch and 4Ch models use the 2Ch and 4Ch slices as an input, respectively. We used the same VGG-inspired architecture for these models. The 2Ch models also have the advantage of being applicable to every patient. Not every patient had a 4Ch slice.

Individually, 2Ch and 4Ch could achieve a similar validation score (**0.0156**) as was achieved by averaging over multiple sax slices. By ensembling only single slice, 2Ch and 4Ch models, we were able to achieve a score of **0.0131** on the public leaderboard.

3.2.3. Patient models

As opposed to single slice models, patient models try to make predictions based on the entire stack of (up to 25) SAX slices. To merge predictions from multiple slices, we designed a layer which combines the areas of consecutive cross-sections of the heart using a truncated cone approximation.

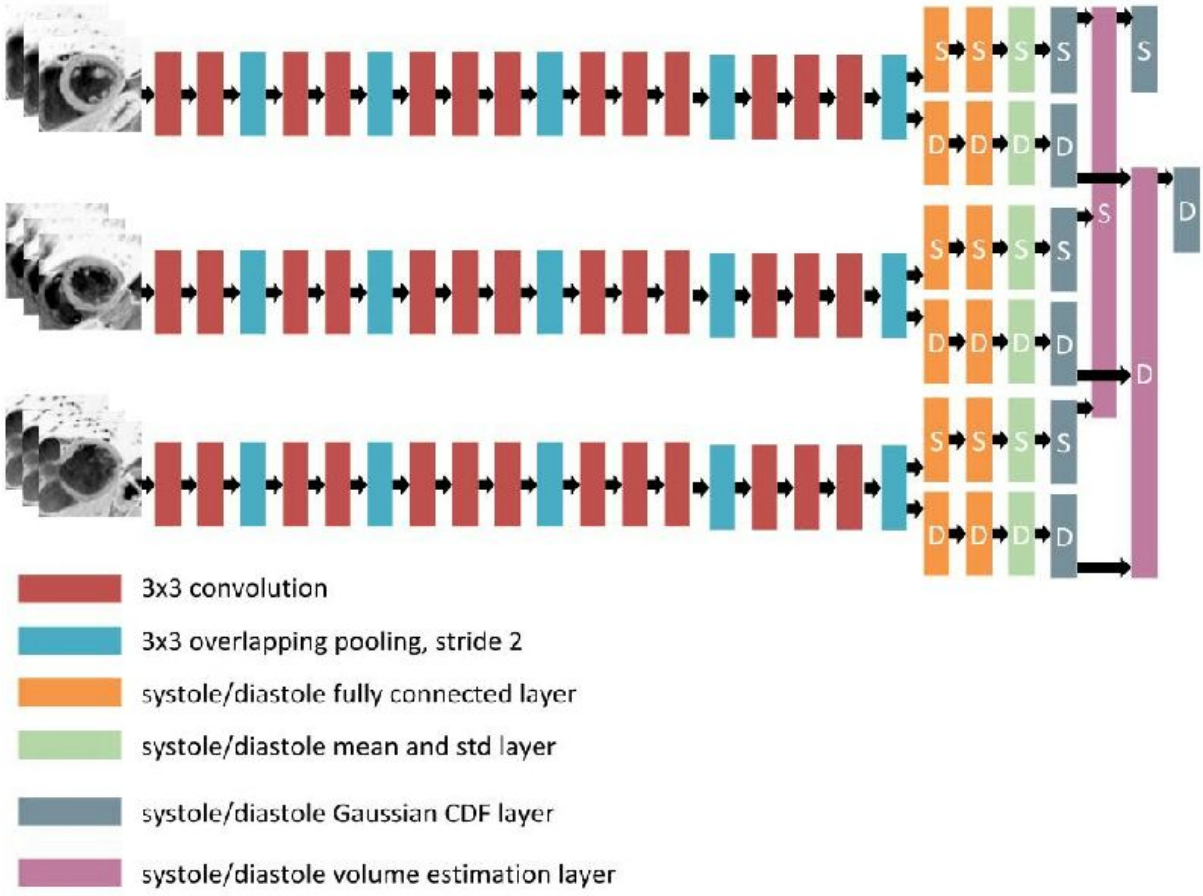
Basically, the slice models have to estimate the area A_i of the cross-section of the heart in a given slice *i*. For each pair of consecutive slices *i* and *i+1*, we estimate the volume of the

heart between them as $V_i = \frac{1}{3}h_{i,i+1}(A_i + \sqrt{A_i A_{i+1}} + A_{i+1})$, where $h_{i,i+1}$ is the distance between the slices. The total volume is then given by $V = \sum_i V_i$.

Assuming that A_i ~ N(μ_i, σ_i^2), then $mean(V_i) = \frac{1}{3}h_{i,i+1}(\mu_i + \sqrt{\mu_i \mu_{i+1}} + \mu_{i+1})$. We found a good approximation of the standard deviation to be $std(V_i) = \frac{1}{3}h_{i,i+1}(\sigma_i + \sigma_{i+1})$ with $std(V) = \sqrt{\sum_i std(V_i)^2}$.

To find the distance $h_{i,i+1}$ between SAX slices, we did not rely on the *SliceThickness* in the metadata. We looked for the two slices of which the center points were furthest apart, drew a line between those center points, and projected every other slice's center point onto this line. This way, we estimated the distance between two slices ourselves. In the case all SAX slices are parallel, the estimation is perfect.

Our best single model achieved a local validation score of **0.0105** using this approach and had the following architecture:



Layer Type	**Size**	**Output shape**
Input layer		(8, 25, 30, 64, 64)*
Convolution	128 filters of 3x3	(8, 25, 128, 64, 64)
Convolution	128 filters of 3x3	(8, 25, 128, 64, 64)
Max pooling		(8, 25, 128, 32, 32)
Convolution	128 filters of 3x3	(8, 25, 128, 32, 32)
Convolution	128 filters of 3x3	(8, 25, 128, 32, 32)
Max pooling		(8, 25, 128, 16, 16)
Convolution	256 filters of 3x3	(8, 25, 256, 16, 16)
Convolution	256 filters of 3x3	(8, 25, 256, 16, 16)
Convolution	256 filters of 3x3	(8, 25, 256, 16, 16)
Max pooling		(8, 25, 256, 8, 8)
Convolution	512 filters of 3x3	(8, 25, 512, 8, 8)
Convolution	512 filters of 3x3	(8, 25, 512, 8, 8)
Convolution	512 filters of 3x3	(8, 25, 512, 8, 8)
Max pooling		(8, 25, 512, 4, 4)
Convolution	512 filters of 3x3	(8, 25, 512, 4, 4)
Convolution	512 filters of 3x3	(8, 25, 512, 4, 4)
Convolution	512 filters of 3x3	(8, 25, 512, 4, 4)
Max pooling		(8, 25, 512, 2, 2)
Fully connected	1024 units	(8, 25, 1024)
Fully connected	1024 units	(8, 25, 1024)
Fully connected	2 units (mu and sigma)	(8, 25, 2)
Volume estimation		(8, 2)
Gaussian CDF		(8, 600)

* The first dimension is the batch size, i.e. the number of patients, the second dimension is the maximum number of slices (if a patient had fewer slices, we padded the inputs, and took this into account in the final layers.)

Most of the time, we did not train patient models from scratch: we initialized patient models with the weights of a trained single slice model. This pre-training decreased the training time and made for more robust models (in the sense that they overfitted less severely).

The architecture we described above was the one that achieved the best results for us. However, during the competition, we trained a lot of diverse networks, some of which also ended up in our final ensemble. Some of the notable things we tried include:

- processing each frame separately, and taking the minimum and maximum at some point in the network to compute systole and diastole
- sharing some of the dense layers between the systole and diastole networks
- using discs to approximate the volume, instead of truncated cones
- cyclic rolling layers (Dieleman et al.)
- leaky RELUs (*y = max(x, a*x) with a = 1/3*)
- maxout units (Goodfellow et al.)

3.3. Training

3.3.1. Miscellaneous

Error function: we found that optimising CRPS directly worked best for us.
Training algorithm: mini-batch gradient descent with Adam (Kingma and Ba) for a fixed number of epochs (usually 100-150) with a discrete decreasing learning rate schedule.
Initialization: we initialised all filters and dense layers orthogonally (Saxe et al.). Biases were initialized to small positive values to have more gradients at the lower layer in the beginning of the optimization. At the Gaussian output layers, we initialized the biases for mu and sigma such that initial predictions of the untrained network would fall in a sensible range.
Regularization: to prevent our models from overfitting we used plenty of data augmentation and a considerable amount of dropout.
Validation: during the first stage of the competition we randomly selected 83 patients for validation. In the second stage, we had 700 patients for training, from which we randomly chose 116 patients for validation.

3.3.2. Selective training and prediction

By looking more closely at the validation scores, we observed that most of the accumulated error was obtained by wrongly predicting only a couple of such outlier cases. At some point, being able to handle only a handful of these meant the difference between a leaderboard score of **0.0148** and **0.0132**.

To mitigate such issues, we set up our framework such that each individual model could choose not to train on or predict a certain patient. For instance, patient models could choose not to predict patients with too few SAX slices, models which use the 4Ch slice would not predict for patients who do not have this slice. We extended this idea further by developing expert models, which only trained and predicted for patients with either a small or a big heart (as determined by the ROI detection step). Further down the pipeline, our ensembling scripts would then take these non-predictions into account.

3.2.3. Test-time augmentation (TTA)

For each individual model, we computed predictions across various augmented versions of the input images and averaged them. We used the acronym TTA to refer to this operation. The augmentation parameters during TTA were the same as during training (except for rescaling). For single slice models random augmentation was sampled per slice and was applied to all 30 images in the MRI sequence. For patient models, augmentation was sampled per patient. This gave an overall improvement of ~0.0004 for patient models.

3.3.4. Ensembling and dealing with outliers

We ended up creating about 250 models throughout the competition. However, we knew that some of these models were not very robust to certain outliers or patients whose ROI we could not accurately detect. We came up with two different ensembling strategies that would deal with these kind of issues.

Our first ensembling technique followed the following steps:

1. For each patient, we select the best way to average over the test time augmentations. Single slice models often preferred a geometric averaging of distributions, whereas in general arithmetic averaging worked better for patient models.
2. We average over the models by calculating each prediction's KL-divergence from the average distribution, and the cross entropy of each single sample of the distribution. This means that models which are further away from the average distribution get more weight (since they are more certain). It also means samples of the distribution closer to the median-value of 0.5 get more weight. Each model also receives a model-specific weight, which is determined by optimizing these weights over the validation set.
3. Since not all models predict all patients, it is possible for a model in the ensemble to not predict a certain patient. In this case, a new ensemble without these models is optimized, especially for this single patient. The method to do this is described in step 2.
4. This ensemble is then used on every patient on the test-set. However, when a certain model's average prediction disagrees too much with the average prediction of all models, the model is thrown out of the ensemble, and a new ensemble is optimized

for this patient, as described in step 2. This meant that about ~75% of all patients received a new, 'personalized' ensemble.

Our second way of ensembling involves comparing an ensemble that is suboptimal, but robust to outliers, to an ensemble that is not robust to them. This approach is especially interesting, since it does not need a validation set to predict the test patients. It follows the following steps:

1. Again, for each patient, we select the best way to average over the test time augmentations again.
2. We combine the models by using a weighted average on the predictions, with the weights summing to one. These weights are determined by optimising them on the validation set. In case not all models provide a prediction for a certain patient, it is dropped for that patient and the weights of the other models are rescaled such that they again sum to one.
3. We combine all 2Ch, 4Ch and slice models in a similar fashion.
4. We detect outliers by finding the patients where the two ensembles (from step 2 and 3) disagree the most. We measure disagreement using CRPS. If the CRPS exceeds a certain threshold for a patient, we assume it to be an outlier. We chose this threshold to be 0.02.
5. We retrain the weights for the first ensemble on the validation set without outliers. Then we choose this ensemble to generate predictions for most of the patients, but choose the second ensemble for the outliers.

Following this approach, we detected three outliers in the test set during phase one of the competition. Closer inspection revealed that for all of them either our ROI detection failed, or the SAX slices were not nicely distributed across the heart. Both ways of ensembling achieved similar scores on the public leaderboard (**0.0108** and **0.0110** respectively).

3.4. Submissions

For the second stage of the competition, we chose to train 44 best models, selected according to our ensembling scripts. For our first submission, we splitted of a new validation set from our training set. The resulting models were combined using our first ensembling strategy. The private leaderboard score of this ensemble was **0.010123**.

For our second submission, we wanted to train our models on the entire training set (i.e. there was no validation split). We then would ensemble them using the second ensembling method. Since we would have no validation set to optimise the weights of the ensemble, we computed the weights by training an ensemble on the models we trained with a validation split (from submission 1), and transferred them over. This submission scored **0.010706** in the final standings.

3.5. Making our solution applicable in practice

This solution can be significantly simplified by removing most of the models from our ensemble. When only ensembling the 5 models `j6_2ch_96mm`, `j6_4ch`, `je_ss_jonisc64small_360`, `meta_gauss_roi10_big_leaky_after_seqshift` and `meta_gauss_roi_zoom_big`, using `merge_predictions.py` (ensembling strategy 1), we got the score of **0.010211** on the private leaderboard, with which we would have achieved a third place.

Reducing the number even more, and keeping only `je_ss_jonisc64small_360` and `meta_gauss_roi_zoom_big`, we achieve a score of **0.010673** on the private leaderboard, which would have given us fourth place. In this case, only the patients where the patient model doesn't create a prediction, are replaced by the predictions of the slice model. Therefore, you do not need a complicated ensemble strategy to achieve this score.

Appendix

A1. Model execution time

The average training time for single slice models and patient models is 8 hours on our machines (see the next section). However, one of the models from our final submission required up to 20 hours to train. Making TTA predictions for the models requires 2-3 hours. Running the ensembling script takes at most 13 hours.

A2. Dependencies

We used Linux machines to develop the code. Most of them ran **Ubuntu 14.04 LTS**, but other distributions should work as well. Some of the code may not run without modifications on Windows or Mac OS (e.g. code using `os.system()` to run commands).

We used **Python 2.7.6** with the following libraries and modules:

- **numpy** 1.10.4 - http://www.numpy.org/
- **scipy** 0.14.0 - http://www.scipy.org/
- **Theano** * - http://deeplearning.net/software/theano/
- **Lasagne** * - http://github.com/benanne/Lasagne
- **scikit-learn** 0.15.2 - http://scikit-learn.org/
- **scikit-image** 0.10.1 - http://scikit-image.org/
- **PyCUDA** 2014.1 - http://mathema.tician.de/software/pycuda/
- **pydicom** - https://github.com/darcymason/pydicom
- **CUDA 7.5 -** https://developer.nvidia.com/cuda-zone
- **cuDNN4 -** https://developer.nvidia.com/cuDNN

*The Theano and Lasagne versions should be the (at the time of writing) development versions. We recommend installing the following git commits, which our code is known to work with: Theano commit ede4874 and Lasagne commit 12db7fb.

All convnets were trained on GPUs. In our lab we had GTX 680's, GTX 980's, Tesla K40's and a Titan X. Although our use of Theano technically allows for our code to be run exclusively on CPUs with minor modifications, this will be much too slow in practice. As a result, a recent NVIDIA GPU (Kepler- or Maxwell-based 600, 700, 800 and 900 series) with 12GB of memory is strongly recommended.

A3. How To Generate the Solution

In this section, we will walk you through how to set up everything from scratch on a Linux machine, run our models, and generate the two final submissions.

During the competition, we worked in two separate frameworks, which we then put into the same repository. The code for the main framework is located in the top folder. The code for the second framework is located in the `ira/` directory.

1. Relevant paths

In this section, we will sum up all the files and paths that are relevant for our code. Some of these paths are configurable in the SETTINGS.json file, in which case we mention the default path.

- `configurations/*.py, ira/configurations/*.py`: configuration files for all network architectures.
- `ira/`: location second framework.
- `ira/sample_submission_validate.csv`: CSV file containing the sample submission.
- `ira/valid_split.pkl`: pickled python dict specifying the train-validation-split (created during the first run of ira/train.py).
- `SETTINGS.json`: JSON file specifying the paths to read from or write to, as well as the submission number.
- `/data/dsb15_final`: default location for the train and test data.
- `/data/dsb15_final/pkl_train/`: default location of the pickled training (i.e. labeled) data.
- `/data/dsb15_final/pkl_train/<PATIENT_ID>/study/*.pkl`: pickle files containing the data of the labeled patient with ID <PATIENT_ID>.
- `/data/dsb15_final/pkl_validate/`: Default location of the pickled test (i.e. unlabeled) data.
- `/data/dsb15_final/pkl_validate/<PATIENT_ID>/study/*.pkl`: The pickle files containing the data of the test patient with ID <PATIENT_ID>.
- `/data/dsb15_final/pkl_train_slice2roi.pkl`: pickle file containing the ROI for each patient in the train set (generated by `pathfinder.py`).
- `/data/dsb15_final/pkl_validate_slice2roi.pkl`: pickle file containing the ROI for each patient in the validation set (generated by `pathfinder.py`).

- `/data/dsb15_final/train.pkl`: pickle file containing the train labels of each patient.
- `/data/dsb15_final/pkl_train_metadata.pkl`: pickle file containing the metadata of each patient in the train set (generated by `generate_metadata_pkl.py`).
- `/data/dsb15_final/pkl_validate_metadata.pkl`: pickle file containing the metadata of each patient in the validation set (generated by `generate_metadata_pkl.py`).

.

2. Setting up

We will assume you use the default SETTINGS.json file. If you want to use different paths, you should change the SETTINGS.json file accordingly

2.1. Download the data and code

Checkout the code from the git repository. Keep track of the folder the code is located in (e.g. by running the command **`export KAGGLE_HEART_CODE='/PATH/TO/CODE'`**)

Download all the provided files from the Kaggle competition web site (and unzip the files were applicable). You should end up with a `train`, `validate` and `test` folder, containing the DICOM files. You should also have `train.csv` and `validate.csv`, containing the labels, and the two sample submissions.

2.2. Combining train and validate

Our code assumes all the labeled data to be in one folder, and generates a validation fold automatically. To combine the train and validate folder, run the following commands:

```
cp -r validate/* train
rm -r validate
tail -n +2 validate.csv >> train.csv
rm validate.csv
mv test validate
```

This last commands will rename the test folder to `validate`, which is what our code assumes the test data folder to be named.

2.3. Everything in it's right place

Move the data and labels to the `/data/dsb15_final` folder. Move and rename the sample submission file to the `$KAGGLE_HEART_CODE/ira` folder.

```
mv                                                    sample_submission_test.csv
$KAGGLE_HEART_CODE/ira/sample_submission_validate.csv
mv train /data/dsb15_final/train
mv validate /data/dsb15_final/validate
mv train.csv /data/dsb15_final/train.csv
```

2.4. Preprocessing

Next, we'll need to preprocess all the data. This includes converting all the DICOM files to pickled numpy arrays, and running the region of interest extraction on them. To do this, run the `pathfinder.py` script.

```
cd $KAGGLE_HEART_CODE/ira
python pathfinder.py
cp pkl_*.pkl /data/dsb15_final
```

This will generate the pickled data in the `/data/dsb15_final/pkl_train` and `/data/dsb15_final/pkl_validate` folders. It will also generate a couple of files containing the results of the ROI extraction. The last command puts a copy of those files in the data folder.

Finally, we need the train labels to be in a pickled format. To do this, run the following commands:

```
cd /data/dsb15_final
python -c "import csv; import numpy; import cPickle;
cPickle.dump(numpy.array(list(csv.reader(open('train.csv','rb'),delimit
er=',')))[1:].astype('float'), open('train.pkl', 'wb'),
cPickle.HIGHEST_PROTOCOL)"
```

3. Training convolutional neural networks with a validation set

Now that everything is set up, we can start training models. Most of the models can be run in parallel, but some models require others to be trained first. In the subsections below, we will specify which models should be run in an order that definitely works.

3.1. Training networks with the main framework

To train a network in the main framework, all you need to do is run the command `python train.py -c <config_name>`. After the model is trained, the script will start generating predictions on the validation and test set automatically.
This command will try to write the model parameters, a log file, the model predictions and a submission to the paths `/mnt/storage/metadata/kaggle-final/train`,

`/mnt/storage/metadata/kaggle-final/logs`, `/mnt/storage/metadata/kaggle-final/predictions` and `/mnt/storage/metadata/kaggle-final/submissions` respectively. Make sure these paths exist, or change the SETTINGS.json file such that existing paths are used. We set those paths to a network disk, such that all results end up on a single disk shared by all machines. Although models are saved regularly, such that training can be restored in case there is a crash of a machine, a regular ethernet network between the machines would do to manage the load generated by the scripts.

```
cd $KAGGLE_HEART_CODE
python train.py -c je_os_fixedaggr_relloc_filtered
python train.py -c je_ss_jonisc64small_360_gauss_longer
python train.py -c j6_2ch_128mm
python train.py -c j6_2ch_96mm
python train.py -c je_ss_jonisc80_framemax
python train.py -c je_os_fixedaggr_rellocframe
python train.py -c j6_2ch_128mm_96
python train.py -c j6_4ch
python train.py -c je_ss_jonisc80_leaky_convroll
python train.py -c je_os_fixedaggr_relloc_filtered_discs
python train.py -c j6_4ch_32mm_specialist
python train.py -c j6_4ch_128mm_specialist
python train.py -c je_ss_jonisc64_leaky_convroll
python train.py -c je_ss_jonisc80small_360_gauss_longer_augzoombright
python train.py -c je_ss_jonisc80_leaky_convroll_augzoombright
python train.py -c je_os_segmentandintegrate_smartsigma_dropout
python train.py -c j6_2ch_128mm_zoom
python train.py -c j6_2ch_128mm_skew
python train.py -c je_ss_jonisc64small_360

python train.py -c je_meta_fixedaggr_filtered
python train.py -c je_meta_fixedaggr_framemax_reg
python train.py -c je_meta_fixedaggr_jsc80leakyconv
python train.py -c je_meta_fixedaggr_joniscale80small_augzoombright
python train.py -c je_meta_fixedaggr_joniscale64small_filtered_longer
python train.py -c je_meta_fixedaggr_jsc80leakyconv_augzoombright_short
python train.py -c je_meta_fixedaggr_joniscale80small_augzoombright_betterdist
```

Most of these networks can be run in parallel (if hardware is available). The only requirement is that the networks whose name starts with `je_ss` have been trained before you start training the models whose name starts with `je_meta.`

Note that `j6_2ch_128mm_zoom` will crash. This is due to a wrong zoom parameter in the model. The zoom used for augmentation in that model is erroneously sampled between [-0.5 and 1.5], instead of [0.5 and 1.5]. When the zoom is sufficiently close to zero - something which should not have been possible - the script crashes. Since we were not allowed to fix parameters in the second round of the competition, we kept this as is. The further scripts were sufficiently robust to deal with the missing predictions from this model.

3.2. Training models with the second framework

The commands below will train single slice and patient models, which were implemented in the second framework. These scripts write log, metadata, prediction and submission files into subfolders of the corresponding main framework folders. For example, log files will be stored in `/mnt/storage/metadata/kaggle-final/logs/ikorshun-paard`, where `ikorshun` is the username and `paard` is the hostname. Only TTA predictions generated by `predict_framework_transfer.py` will be stored in `/mnt/storage/metadata/kaggle-final/predictions`

```
cd $KAGGLE_HEART_CODE/ira

python train.py gauss_roi10_maxout_seqshift_96
python train_meta.py meta_gauss_roi10_maxout_seqshift_96
python predict_framework_transfer.py gauss_roi10_maxout_seqshift_96 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_maxout_seqshift_96 50 arithmetic

python train.py gauss_roi10_big_leaky_after_seqshift
python train_meta.py meta_gauss_roi10_big_leaky_after_seqshift
python predict_framework_transfer.py gauss_roi10_big_leaky_after_seqshift 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_big_leaky_after_seqshift 50 arithmetic

python train.py gauss_roi_zoom_big
python train_meta.py meta_gauss_roi_zoom_big
python predict_framework_transfer.py gauss_roi_zoom_big 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_big 50 arithmetic

python train.py gauss_roi10_zoom_mask_leaky_after
python train_meta.py meta_gauss_roi10_zoom_mask_leaky_after
python predict_framework_transfer.py gauss_roi10_zoom_mask_leaky_after 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_zoom_mask_leaky_after 50 arithmetic

python train.py gauss_roi10_maxout
python train_meta.py meta_gauss_roi10_maxout
python predict_framework_transfer.py gauss_roi10_maxout 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_maxout 50 arithmetic

python train.py gauss_roi_zoom_mask_leaky_after
python train_meta.py meta_gauss_roi_zoom_mask_leaky_after
python predict_framework_transfer.py gauss_roi_zoom_mask_leaky_after 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_mask_leaky_after 50 arithmetic

python train.py gauss_roi_zoom_mask_leaky
python train_meta.py meta_gauss_roi_zoom_mask_leaky
python predict_framework_transfer.py gauss_roi_zoom_mask_leaky 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_mask_leaky 50 arithmetic

python train.py gauss_roi_zoom
python train_meta.py meta_gauss_roi_zoom
python predict_framework_transfer.py gauss_roi_zoom 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom 50 arithmetic
```

```
python train.py ch2_zoom_leaky_after_maxout
python predict_framework_transfer.py ch2_zoom_leaky_after_maxout 50 arithmetic

python train.py ch2_zoom_leaky_after_nomask
python predict_framework_transfer.py ch2_zoom_leaky_after_nomask 50 arithmetic
```

Again, some parallelism is possible here. We grouped together the commands that should be run consecutively, but the different groups can be run in parallel.

4. Ensembling the trained networks (part 1)

At this point, you should have a bunch of prediction files in the predictions (`/mnt/storage/metadata/kaggle-final/predictions`) folder, one for each network. To generate an ensemble from these, and produce the first submission, you should run the script `merge_predictions.py`.

```
cd $KAGGLE_HEART_CODE
python merge_predictions.py
```

This script will make a weighted average of the predictions of all the generated predictions. The weights are obtained by minimising the loss on the validation set. Consecutively, for each test patient, the script will remove any models that disagree too much with the others for that specific patient, and re-optimise the weights.

Once the script finishes, it will write the submission file to the submissions directory (`/mnt/storage/metadata/kaggle-final/submissions`). It will also print the destination path to the console. This submission is the one with which we achieved second place on the private leaderboard.

To generate the second submission, we need to retrain every network on the full training set. Then, we will create an ensemble out of them using the `merge_predictions_jeroen.py` script. To be able to do this though, we first need to compute the ensemble weights on the current networks and store them. Do this by running the following commands:

```
cd $KAGGLE_HEART_CODE
python merge_predictions_jeroen.py
```

The ensemble weights will be exported to the appropriate folder (`/mnt/storage/metadata/kaggle-final/ensemble-weights`). The full path will also be printed to console. Keep track of this file, e.g. by running the command **`export KAGGLE_WEIGHTS_FILE='/PATH/TO/WEIGHTS/FILE'`** . You are now ready to set up for the second submission.

5. Setting up for the second submission

Before starting to retrain your models, you will want to change the target directories, which the scripts will write to, to new empty ones. Otherwise, the models from the first and second submission will be confused with each other. This can be done by changing the **SETTINGS.json** file. In the same file, you will also need to set the SUBMISSION_NR to 2. Alternatively, you can simply copy over the settings file we prepared:

```
cd $KAGGLE_HEART_CODE
cp settings_files/SETTINGS_SUB_2.json SETTINGS.json
```

Again, make sure all the new target directories exist.

6. Training convolutional neural networks with a validation set

You are now ready to retrain all the networks on the full training set. The procedure is the entirely the same as previously.

```
cd $KAGGLE_HEART_CODE
python train.py -c je_os_fixedaggr_relloc_filtered
python train.py -c je_ss_jonisc64small_360_gauss_longer
python train.py -c j6_2ch_128mm
python train.py -c j6_2ch_96mm
python train.py -c je_ss_jonisc80_framemax
python train.py -c je_os_fixedaggr_rellocframe
python train.py -c j6_2ch_128mm_96
python train.py -c j6_4ch
python train.py -c je_ss_jonisc80_leaky_convroll
python train.py -c je_os_fixedaggr_relloc_filtered_discs
python train.py -c j6_4ch_32mm_specialist
python train.py -c j6_4ch_128mm_specialist
python train.py -c je_ss_jonisc64_leaky_convroll
python train.py -c je_ss_jonisc80small_360_gauss_longer_augzoombright
python train.py -c je_ss_jonisc80_leaky_convroll_augzoombright
python train.py -c je_os_segmentandintegrate_smartsigma_dropout
python train.py -c j6_2ch_128mm_zoom
python train.py -c j6_2ch_128mm_skew
python train.py -c je_ss_jonisc64small_360

python train.py -c je_meta_fixedaggr_filtered
python train.py -c je_meta_fixedaggr_framemax_reg
python train.py -c je_meta_fixedaggr_jsc80leakyconv
python train.py -c je_meta_fixedaggr_joniscale80small_augzoombright
python train.py -c je_meta_fixedaggr_joniscale64small_filtered_longer
python train.py -c je_meta_fixedaggr_jsc80leakyconv_augzoombright_short
```

```
python train.py -c je_meta_fixedaggr_joniscale80small_augzoombright_betterdist
```

```
cd $KAGGLE_HEART_CODE/ira

python train.py gauss_roi10_maxout_seqshift_96
python train_meta.py meta_gauss_roi10_maxout_seqshift_96
python predict_framework_transfer.py gauss_roi10_maxout_seqshift_96 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_maxout_seqshift_96 50 arithmetic

python train.py gauss_roi10_big_leaky_after_seqshift
python train_meta.py meta_gauss_roi10_big_leaky_after_seqshift
python predict_framework_transfer.py gauss_roi10_big_leaky_after_seqshift 50 arithmetic
python    predict_framework_transfer.py    meta_gauss_roi10_big_leaky_after_seqshift    50
arithmetic

python train.py gauss_roi_zoom_big
python train_meta.py meta_gauss_roi_zoom_big
python predict_framework_transfer.py gauss_roi_zoom_big 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_big 50 arithmetic

python train.py gauss_roi10_zoom_mask_leaky_after
python train_meta.py meta_gauss_roi10_zoom_mask_leaky_after
python predict_framework_transfer.py gauss_roi10_zoom_mask_leaky_after 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_zoom_mask_leaky_after 50 arithmetic

python train.py gauss_roi10_maxout
python train_meta.py meta_gauss_roi10_maxout
python predict_framework_transfer.py gauss_roi10_maxout 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi10_maxout 50 arithmetic

python train.py gauss_roi_zoom_mask_leaky_after
python train_meta.py meta_gauss_roi_zoom_mask_leaky_after
python predict_framework_transfer.py gauss_roi_zoom_mask_leaky_after 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_mask_leaky_after 50 arithmetic

python train.py gauss_roi_zoom_mask_leaky
python train_meta.py meta_gauss_roi_zoom_mask_leaky
python predict_framework_transfer.py gauss_roi_zoom_mask_leaky 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom_mask_leaky 50 arithmetic

python train.py gauss_roi_zoom
python train_meta.py meta_gauss_roi_zoom
python predict_framework_transfer.py gauss_roi_zoom 50 arithmetic
python predict_framework_transfer.py meta_gauss_roi_zoom 50 arithmetic

python train.py ch2_zoom_leaky_after_maxout
python predict_framework_transfer.py ch2_zoom_leaky_after_maxout 50 arithmetic

python train.py ch2_zoom_leaky_after_nomask
python predict_framework_transfer.py ch2_zoom_leaky_after_nomask 50 arithmetic
```

Again, most networks can be run in parallel.

7. Ensembling the trained networks (part 2)

Once all networks have been trained and the predictions generated, we can make an ensemble out of them using the weights we computed earlier:

```
cd $KAGGLE_HEART_CODE
python merge_predictions_jeroen.py $KAGGLE_WEIGHTS_FILE
```

This script will write a couple of submission to the submissions directory, and write the paths to those to the console. The submission you want is called `ensemble_final.XXXX.XX.csv`, 'X' being some number.

For us, this second submission scored a little bit worse (namely **0.010706** on the private leaderboard) than the first one.