Table of Contents

[File Structure 3](#_Toc23861212)

[Step 1: Handling the dataset 5](#_Toc23861213)

[Step 2: Image Preprocessing 6](#_Toc23861214)

[Custom Image Preprocessing: 7](#_Toc23861215)

[Custom Crop function: 7](#_Toc23861216)

[Image Preprocessing Future Functionality 10](#_Toc23861217)

[Step 3: Model Training and Tuning 11](#_Toc23861218)

[Hyperparameters 14](#_Toc23861219)

[Training Techniques 17](#_Toc23861220)

[Review Training Session 18](#_Toc23861221)

[Model Training Future Features 22](#_Toc23861222)

[Step 4: Model Testing 23](#_Toc23861223)

[Statistical Analysis 23](#_Toc23861224)

[Future Improvements 25](#_Toc23861225)

Main components:

1. Data Preprocessing
2. Model Training
3. Testing /Model Tuning

Execution flow of the program:

[**Step 1**](#_Step_1:_Handling): User places DICOM images in “./data/dicom-images-train” and “./data/dicom-images-test-unofficial”.

[**Step 2**](#_Step_2:_Image)**:** User performs image preprocessing through Main.py’s menu.

[**Step 3**](#_Step_3:_Model): User trains different types of models with an assortment of hyperparameters through Main.py’s menu.

[**Step 4**](#_Step_4:_Model): User tests their best models on the test dataset through Main.py’s menu.

# File Structure

**Bold** are files.

* data
  + dicom-images-train
  + dicom-images-train-filtered
  + dicom-images-test-unofficial
  + dicom-images-test-unofficial-filtered
  + **train\_labels.csv**
  + **test\_labels.csv**
* Documentation
  + **This document**, and other documentation
* documents
  + **Miscellaneous documents** created by the user to aid in the research or model training
* trained\_models
  + Stores temporary model .h5 files
  + **hyperparameters.json**
* training\_sessions
  + binary
    - cnn
      * 2019-10-29
        + 1

**.h5 model**

**hyperparameters.json**

**periphery\_data.json**

**train\_stats\_x\_of\_n.json**

**…**

**validation\_stats\_x\_of\_n.json**

**…**

* + - * + 2

**…**

* + - * + ….
      * …
    - unet
      * …
    - …
  + segment
    - …
* **[Python files]**

Files and their purpose:

|  |  |
| --- | --- |
| Main.py | Main class handling the menu and user input, and calls other classes as needed. |
| Classifier.py | Houses the classes that handle model creation and model training. |
| DataGenerator.py | Houses the class that Keras uses to read input images from disk in batches. Useful for large datasets where the whole training dataset cannot be loaded into memory at once. |
| DataHandler.py | Handles data retrieval and writing to disk. |
| DICOM\_reader.py | Houses the class that handles DICOM images, whether that be read, plot, display, or modify. |
| ImagePreprocessor.py | Handles the preprocessing of DICOM images. Includes cropping, edge detection, etc. |
| Tester.py | Handles the testing of previously trained models located in ./training\_sessions. |

# Step 1: Handling the dataset

The user currently determines how to split the dataset into train and test datasets, because there’s currently no implemented way to do it programmatically. The test dataset is populated by 20% of the initial dataset. The train dataset is populated by the other 80%.

Location to place the training dataset DICOM files: “./data/dicom-images-train”.

Location to place the testing dataset DICOM files: “./data/dicom-images-test-unofficial”.

The user can place the DICOM files in folders if that’s their preference, but the code retrieves all DICOM images in the aforementioned locations, and considers them all equal.

Dataset labels are annotations in run-length-encoded (RLE) masks. This is the only target label format supported, currently.

EX: 378999 2 1018 8 1013 13 1009 15 1007 17 1006 17 1005 19 1002 21 1001 22 1001 22 1000 23 1000 23 1000 23 998 25 997 26 996 27 996 27 995 28 994 29 994 29 993 30 993 30 992 31 992 31 992 31 991 33 990 33 990 33 990 33 990 33 990 33 990 33 991 32 991 32 991 32 991 32 990 33 990 33 990 33 989 34 989 34 988 35 988 35 988 35 989 34 989 34 989 34 989 34 989 34 989 34 989 34 990 34 989 34 990 33 991 32 991 32 992 31 992 31 992 31 993 31 992 31 993 30 993 31 993 30 993 30 994 30 993 30 994 30 994 29 994 29 995 29 994 29 995 29 994 29 995 29 995 29 994 29 994 30 994 30 993 30 993 31 993 31 993 30 994 30 994 30 991 32 990 34 990 33 991 33 990 33 991 33 991 33 990 34 990 33 991 33 990 34 990 34 990 34 990 33 991 33 990 34 990 34 990 34 990 34 990 34 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 35 989 36 988 36 988 36 988 37 987 37 987 37 987 38 986 38 986 38 986 38 986 39 985 39 985 39 985 40 984 40 984 41 983 41 983 42 982 42 982 42 982 43 981 43 982 42 982 43 981 43 981 44 980 44 981 43 981 44 980 45 980 44 980 45 979 45 980 45 979 46 978 46 978 47 977 47 978 47 977 48 976 49 975 50 974 50 975 50 974 51 974 50 974 51 974 50 974 51 973 52 973 51 973 52 973 51 973 51 974 50 974 51 974 50 974 50 975 50 974 50 974 50 975 49 976 48 977 47 978 46 979 45 980 43 982 42 983 39 986 37 988 34 991 32 993 29 997 25 1002 21 1006 16

The dataset labels shall be placed in csv files named “train\_labels.csv” and “test\_labels.csv”. These shall be located in “./data”. Each file will have two columns, one for the name of the image file without the extension, and one column for the RLE masks, where -1 means no mask.

|  |  |
| --- | --- |
| ImageId | EncodedPixels |
| 1.2.276.0.7230010.3.1.4.8323329.6904.1517875201.850819 | -1 |
| 1.2.276.0.7230010.3.1.4.8323329.13666.1517875247.117800 | [RLE mask] |
| 1.2.276.0.7230010.3.1.4.8323329.11028.1517875229.983789 | -1 |
| 1.2.276.0.7230010.3.1.4.8323329.10366.1517875223.393986 | [RLE mask] |
| 1.2.276.0.7230010.3.1.4.8323329.10016.1517875220.992175 | [RLE mask] |
| 1.2.276.0.7230010.3.1.4.8323329.11444.1517875232.977506 | -1 |
| 1.2.276.0.7230010.3.1.4.8323329.32219.1517875159.70802 | -1 |

# Step 2: Image Preprocessing

In this step, the DICOM images in either the training or testing dataset will have preprocessing performed on them, and then saved as .png files in another folder.

Current image preprocessing techniques available:

* Crop
* Gaussian Blur
* Gradient detection
* Threshold
* Brightening
* Canny Edge Detection
* Custom edge filter

Main Menu:

---- Pneumothorax Classifier ----

-- Menu --

1) Preprocess scans

2) Train/Test model

3) View Train/Test results

0) Quit

Choice:

The user will choose 1 to perform image preprocessing.

-- Preprocessor Menu --

Which dataset portion to preprocess?

1) Training dataset

2) Testing dataset

Choice:

The user can specify whether to perform preprocessing on the training dataset images or testing dataset images.

After specifying, the user will be presented with another menu:

Wish to replace existing preprocessed images? (y/n):

If yes, all existing preprocessed images will be replaced.

The preprocessing step is now underway, with the custom preprocess() function in the class specific to the project.

# Custom Image Preprocessing:

1. Custom crop of the radiograph to only include relevant body part
2. resizes to 512x512 pixels
3. Performs Canny Edge Detection, which currently executes
   1. Perform threshold calculation
   2. Subtracts the threshold from the image
4. If the mean of current image is less than a certain intensity, a variable is modified and the image has Canny Edge Detection performed again.
5. If the intensity of the original DICOM image was less than a certain intensity, then the image has its contrast increased.

The final result is a 2D array of pixel intensities, which is saved as a png file in either “./data/dicom-images-train-filtered” or “./data/dicom-images-test-unofficial-filtered”.

Depending on the computer, bulk preprocessing 10,000 images might take 30 minutes+.

# Custom Crop function:

Detecting anomalies in the lungs will be used as an example for a custom crop function.

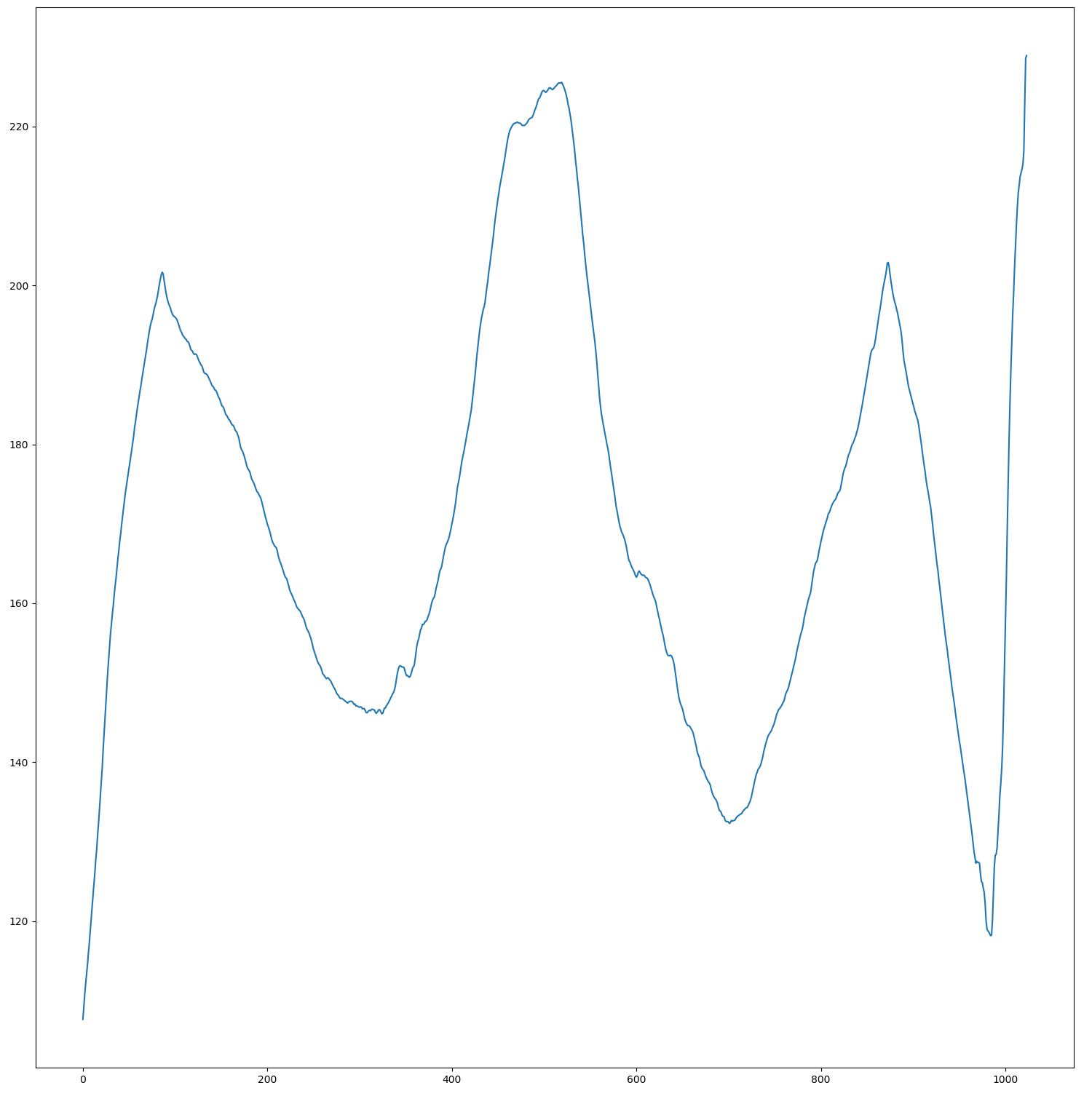
A chest radiograph may include a portion of someone’s neck, shoulders, arms, abdomen, and even be misaligned massively. All of these are unnecessary in capturing anomalies in the lungs, so we want to crop our radiographs to include contain the rib cage. This can be done by programmatically or manually.

Programmatically, we first crop the sides of the image so that the edge of the right ribcage is touching the left side of the image, and the edge of the left ribcage is touching the right side of the image.

Determining where the sides of the ribcage are, can be done by

1. Average each column’s intensity.
2. Run through a local maximum filter to get local maxima of the intensities.
3. Use binary erosion to determine local peaks.
4. Return list of indices pointing to local and global peaks in intensity.
5. Get index closest to middle of image.
6. Get closest peaks to that index.

An example of column intensities:



In this example of the ribcage, the maximum in the middle of the image is the spine, since the spine is bright in comparison to the lungs. The minimums on both sides of the middle maximum, are the lungs themselves. The maxima towards the edges can either be the person’s arm(s), light leakage in the scan, or the edge of the ribcage.

The list of indices of max peak intensities will look like this for the above example:

[98, 515, 866, 1024]

Once we have a list of maxima intensities, we determine the middle of the ribcage by getting the middle maximum, which is usually the spine. The sides of the ribcage are usually the maxima closest to the spine.

In our example, 515 is denoted as the spine since it’s closest to the middle of the image (1024/2 = 512). Because 515 is considered the spine, 98 and 866 are considered the edge of the ribcages. Our image will then be cropped from 98 to 866 on the sides.

Now that we know where to crop horizontally, we determine where to crop vertically. This is so that we only include from the top of the ribcage to the bottom of the ribcage, and not include the person’s neck, abdomen, etc.

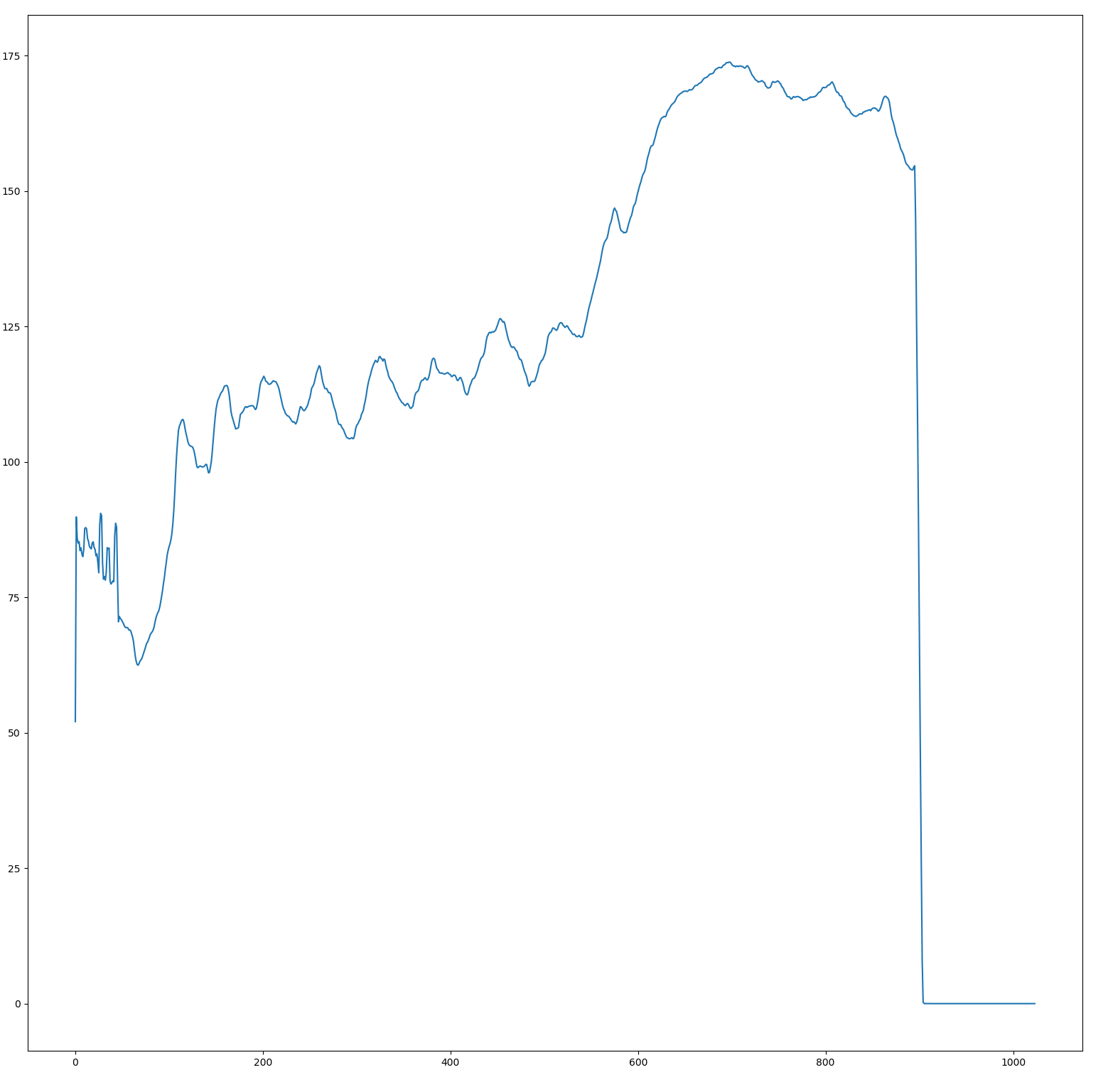
A similar process is performed, but is a different “algorithm”.

Programmatically, we crop the top and bottom of the image so that the top of the ribcage is touching the top part of the image, and the bottom of the ribcage is touching the bottom part of the image.

Determining where the top and bottom of the ribcage are, can be done by

1. Average each row’s intensity.
2. Run through a local maximum filter to get local maxima of the intensities.
3. Run through a local minimum filter to get local minima of the intensities.
4. Use binary erosion to determine local peaks.
5. Return list of indices pointing to local and global minima in intensity.
6. Find local minimum closest to middle of the image.
7. Get index of nearest maxima to the middle minimum.

An example of row intensities:



In this example of the ribcage, the

The list of indices of maxima intensities will look like this for the above example:

[5, 690, 1024]

The list of indices of minima intensities will look like this for the above example:

[0, 89, 307, 1024]

Now we determine the vertical middle of the ribcage by getting the area of least intensity in the middle of the image. This will be considered the middle of the lungs, since they have the lowest pixel intensity when they’re widest.

In our example, 307 is denoted as the vertical middle of the ribcage since it’s closest to the middle of the image (1024/2 = 512, which is closer to 307 than 1024). Because 307 is considered the middle of the ribcage vertically, the maxima closest to this middle minimum, are considered the top and bottom of the ribcage; 5 and 690 respectively. Our image will then be cropped from row of pixels 5 to row of pixels 690.

This method is not perfect, since many scans can have imperfections and bright spots leading to the cropping function to be too aggressive or too passive. It is meant to mass crop thousands of images, then for a human to review the images and manually fix any errors.

# Image Preprocessing Future Functionality

Discussed further in the [Future Functionality](#_Future_Improvements) section.

Future functionality includes:

* User being able to specify which preprocessing techniques to perform.
* Custom cropping function for radiographs of other body parts.

# Step 3: Model Training and Tuning

In this step, the user trains multiple models with multiple architectures using multiple combinations of hyperparameters and multiple ensembling techniques. This is all facilitated with a central menu system, avoiding hard coded values and for streamlining hyperparameter searching.

Current model architectures available:

* CNN
* U-net

Current classification types available:

* Binary classification
* Segmentation

Current model training and tuning techniques available (More info in [Training Techniques](#_Training_Techniques)):

* Split of 50% positive and 50% negative for training dataset
* Early Stopping
* Early Saving
* Weighted Average Ensemble
* Bootstrap Aggregation (Bagging)
* Resampling Ensemble
* K-fold Cross Validation

Main Menu:

---- Pneumothorax Classifier ----

-- Menu --

1) Preprocess scans

2) Train/Test model

3) View Train/Test results

0) Quit

Choice:

The user will choose 2 to train models

Menu presented:

-- Classifier Menu --

Classification type:

1) Binary (Predicting positive or negative)

2) Segmentation (Predicting segments)

0) Quit

Choice:

The user can specify what type of classification they want to perform, whether it’s performing a binary prediction of yes or no, or a segmentation prediction where the area of interested is predicted in the form of a pixel mask.

After specifying, the user will be presented with another menu:

Model building step:

1) Train

2) Test

0) Quit

Choice:

The user can choose whether they want to train a new model with hyperparameters they will soon specify, or test an existing model that has been saved as a training session in “./training\_sessions/…” (More info in [Step 4: Model Testing](#_Step_4:_Model))

If the user chooses to train a new model, they will be presented with this menu:

Model architecture:

1) CNN

2) U-net

0) Quit

Choice:

The user can choose what model architecture they wish to train. A u-net is made of convolutional layers, but has a distinct U shape due to its filtering and subsequent upscaling. More model architectures will be supposed in the future.

Matter the choice, the user will be presented with this next menu where they can modify the hyperparameters that correspond to the classification type they selected and model architecture:

Hyperparameters:

{

"activation\_regularization": 0.0,

"augmented": true,

"bagging\_num\_models": 3,

"balanced": true,

"batch\_normalization": false,

"batch\_size": 10,

"conv\_activation": "relu",

"conv\_layer\_size": 16,

"dataset\_size": 1000,

"dense\_activation": "relu",

"dropout": 0.1,

"epochs": 30,

"filter\_size": 3,

"kfold\_cross\_validation\_k\_folds": 5,

"last\_layer\_size": 32,

"loss": "binary\_crossentropy",

"noise\_in\_hidden\_layer": false,

"noise\_std": 0.0,

"num\_conv\_layers": 4,

"optimizer": "adam",

"output\_activation": "sigmoid",

"pool\_size": 3,

"resampling\_ensemble\_n\_splits": 0,

"train\_ratio": 0.7,

"val\_ratio": 0.2,

"weight\_limit": 2,

"weight\_regularization": 0.0001,

"weighted\_avg\_ensemble\_num\_models": 3

}

Modify? (y/n):

More hyperparameters will be modifiable in the future as more architectures and model training/tuning techniques are added. There will be hyperparameters that are specific to certain classification types or model architectures. However, no matter the training technique specified, all training techniques hyperparameters will be listed.

Full details on each hyperparameter’s meaning are located in [Hyperparameters](#_Hyperparameters) section.

If the user chooses “y” to modify any of the hyperparameters, they will be presented with a prompt for the name of the hyperparameter variable, and the new value to give it. The new value has to be the same data type as the old value. So “augment” can either be changed to true or false, not 0 or 1. “dropout” can only be changed to a float, like 0.3, not an int or a string.

After they user is done modifying as many hyperparameter variables as they would like, they are then presented with this final menu:

Model training type:

1) Standard

2) Resample Ensembling

3) K-fold cross validation

4) Model averaging

5) Bagging (Bootstrapping Aggregation)

0) Quit

Choice:

The user can now specify what type of training they want to perform. A detailed explanation on the different training techniques are located in section [Training Techniques](#_Training_Techniques)

At this point, the training is underway. The results will be printed on the screen, and saved to disk under a new training session folder. The results are calculated on a validation dataset that is 20% of the dataset size specified in the hyperparameters.

# Hyperparameters

|  |  |  |
| --- | --- | --- |
| **Var name** | **Datatype** | **Description** |
| Activation\_regularization | float | The l1 regularization value that is passed into keras’ “activity\_regularizer” parameter, used to reduce overfitting by performing l1 regularization on hidden weights. |
| Augmented | boolean | true if the user wants the radiographs in the training dataset to be augmented by appending the same training images, but altered. Alterations include rotation, stretching in horizontal or vertical direction, shearing, and mirroring. Purpose is to better train the model for out-of-sample images and reduce overfitting. |
| Bagging\_num\_models | int | The number of models to train for the Bagging ensembling technique. |
| Balanced | boolean | True if the user wants the training dataset to have a balanced amount of positive and negative ground truths. A 50% split between the two no matter the original ratio of positive and negative ground truths. Models train better when they have an ample amount of positive signals during training, instead of the negative signals overpowering the sparse positive ones. |
| Batch\_normalization | boolean | True if the user wants batch normalization to be performed inbetween Conv layers while training. Supposed to speed up training and reduce overfitting. |
| Batch\_size | int | The number of items per batch. The higher the batch size, the higher the memory usage since more items need to be in memory, but the quicker the training. Higher batch sizes tend to also reduce overfitting by reducing effective learning rate without literally reducing the learning rate. |
| Conv\_activation | string | The name of the activation function to use in the input and hidden convolution layers. Options include keras’ supported activation functions: relu, selu, elu, softmax, softplus, softsign, tanh, sigmoid, hard\_sigmoid, exponential, and linear. (https://keras.io/activations/). Relu is typically ideal. |
| Conv\_layer\_size | int | The size of the convolutional layer filter, which is the size of the output from that layer. The larger the filter size, the larger the features that can be detected in the image, but also higher memory requirement due to the increase in trainable parameters of the overall model. |
| Dataset\_size | int | The size of the data to train and validate on. From this dataset, 80% will be used for training and 20% will be used for post-training validation to give the results. From the 80% that will be used for training, 80% of that (64% of original) will be used for actual training, and 20% of that (16% of original), will be used for validation during training. This during-training validation dataset will be used to determine [Early Stopping](#_Training_Techniques) and [Early Saving](#_Training_Techniques). |
| Dense\_activation | string | The activation function for the dense layers in regular CNN model architecture. Options are same as in Conv\_activation: relu, selu, elu, softmax, softplus, softsign, tanh, sigmoid, hard\_sigmoid, exponential, and linear. (https://keras.io/activations/). Relu is typically ideal. |
| Depth | int | The depth of the u-net for the u-net model architecture. A depth of 1 will have the u-net only downscale and upscale once. A depth of 2 will have the u-net downscale twice and then upscale twice. The downscaling and upscaling amount depends on filter\_size. The larger the depth (up to a certain point), the stronger the feature recognition and therefore better the training. Also, the larger the depth, the higher memory requirement due to the increase in trainable parameters of the overall model. |
| Dropout | float | The percentage of weights in-between each layer to drop and not propagate to the next layer. The higher the dropout rate, the higher the likelihood for each weight to be “dropped”. Purpose is to reduce overfitting. A value of 0.2 - 0.5 is typically ideal. |
| Epochs | int | The number of training “cycles” to perform to train the model. Each epoch represents a full cycle of the whole training dataset being trained on and then backpropagation updating the neural network weights. The more epochs, the better a model trains to fit the training dataset. To a certain point, increasing epochs increases overfitting, and decreasing epochs increases underfitting. See [Early Stopping](#_Training_Techniques). |
| Filter\_size | int | The height and width of the convolution window. Also referred to as “kernel\_size” in Keras. The larger the height and width, the larger the convolution window, and more information that transfers between layers. This has a tradeoff of increasing the number of trainable parameters. A value of 3-5 is typically ideal. |
| Kfold\_Cross\_validation\_k\_folds | int | The number of folds to use in [k-fold cross validation](#_Training_Techniques). If equal to 2, then half the dataset will be used for training and half for validation. If set to 10, then 10% of the dataset will be used for validation, and 90% for training. However, this 10% portion changes to the next 10% portion, and so on until all data in the dataset has been used or validation. The higher the number of folds, the better the results from K-fold cross validation will represent the actual performance of the model. |
| Last\_layer\_size | int | The size of the layer directly preceding the output layer. For binary classification, this is a dense layer, and for segmentation classification, it’s a convolution layer. The larger the layer, the more trainable parameters in the overall model. |
| Loss | string | The loss function that will be used for calculating error and backpropagation. Options include keras’ supposed loss functions: mean\_squared\_error (mse), mean\_absolute\_error (mae), mean\_absolute\_percentage\_error, crossentropy , etc. (https://keras.io/losses/). |
| Noise\_std | float | Standard deviation of the noise distribution for the Gaussian noise layer. Gaussian noise layer is a regularization layer designed to mitigate overfitting. A large Noise\_std value, the larger the distribution of Gaussian noise values, increasing the noise in the system. 0.01 is typically ideal, with 0.0 meaning no noise regularization. |
| Num\_conv\_layers | int | Specifies the total number of convolutional layers to have in the CNN model architecture. The more convolutional layers you have, the more the model fits to the training dataset. This can fix an underfitting problem, but can easily lead to an overfitting problem. |
| Optimizer | string | The optimizing function is the learning algorithm, the algorithm used to perform gradient descent or learn by lowering loss value. Options include Keras’ supposed optimizers: sgd, rmsprop, adagrad, adadelta, ada, adamax, and nadam. (https://keras.io/optimizers/). |
| Output\_activation | string | The name of the activation function to use in the output/target layer. Options include keras’ supported activation functions: relu, selu, elu, softmax, softplus, softsign, tanh, sigmoid, hard\_sigmoid, exponential, and linear. (https://keras.io/activations/). Sigmoid is typically ideal. |
| Pool\_size | int | Size of the pool filter, which is the amount to downsample. Purpose is to reduce the spatial size of the image, thereby reducing the number of trainable parameters. A value of 2-4 is typically ideal. |
| Resampling\_ensemble\_n\_splits | int | The number of splits to perform on the dataset for [Resampling Ensemble](#_Training_Techniques). The larger the n\_splits, the less data is used to train for each model, but the more models that have been trained on a portion of the dataset. 5 is typically ideal. |
| Train\_ratio | float | The percentage of the dataset to be used for training. If 0.8, 80% of the dataset will be used for training. A value of 0.6-0.9 is typically ideal. |
| Val\_ratio | float | The percentage of the dataset to be used for validation. If 0.2, 20% of the dataset will be used for validation. A value of 0.1-0.3 is typically ideal. Val\_ratio+Train\_ratio should not exceed 1.0. |
| Weight\_limit | int | The max value to constrain the weights in the neural network. Purpose is to avoid exploding gradients during training. Referred to as “kernel\_constraint” in Keras documentation. A value of 2-4 is typically ideal. |
| Weight\_regularization | float | Regularization value used for l2 regularization on weights. Purpose is to avoid overfitting during training. Referred to as “kernel\_regularizer” in Keras documentation. A value of 0.0001 – 0.01 is typically ideal. |
| Weighted\_avg\_ensemble\_num\_models | int | The number of models to train for [Averaging Model Ensemble](#_Training_Techniques). The more models to train, the better the weighted average of the system. A value of 3-10 is typically ideal. |

# Training Techniques

* Split of 50% positive and 50% negative for training dataset
* Early Stopping
* Early Saving
* Weighted Average Ensemble
* Bootstrap Aggregation (Bagging)
* Resampling Ensemble
* K-fold Cross Validation

Balanced Split: Transforming the dataset to have 50% positive signals and 50% negative signals, allows any model trained to have a better sample of positive signals than with the original dataset. Especially if the dataset is extremely unbalanced, like if the original dataset only contains 5% positive signals.

Early Stopping: When training a neural network model, the loss on the training set and validation set typically go down together. At some point, the loss on the training set will still be going down, but the loss on the validation set will start to increase. This loss will keep increasing as the model keeps training, because at this point, it’s now overfitting to the training set. We want to stop training when validation loss starts increasing. Early Stopping stops training if the validation loss increases for a certain number of epochs. Early Stopping stops overfitting for the most part.

Early Saving: Saves model with lowest validation loss during training. Early Saving and Early Stopping are similar, in that they reduce overfitting. When training a neural network model, the loss on the training set and validation set typically go down together. At some point, the loss on the training set will still be going down, but the loss on the validation set will start to increase. This loss will keep increasing as the model keeps training, because at this point, it’s now overfitting to the training set. Early Saving continuously saves the model with the lowest yet validation loss, therefore instead of the training session returning the final model, it returns the model that saw the lowest validation loss during training. Early Saving reduces overfitting and increases overall accuracy.

Reducing dropout: Some papers have shown that reducing the dropout ratio in later layers of the model, will increase accuracy. This follows the same philosophy in artificial intelligence where the best decision models have a high rate of exploration when first starting, then get greedier as their learning progresses. In the sense of neural networks, we want our dropout to be larger at first since the first layer or two are “exploring” weights, then later in the neural network, we want out dropout to be smaller or 0 so that we really focus the learning on the learned weights.

Weighted Average Ensemble: The Wisdom of Crowds is an idea that the collective wisdom of a crowd of people will be higher than each individual person by themselves. That same idea is applied here, where an ensemble of similar models will produce more accurate predictions than each of those models individually. But some models in that ensemble are more accurate than others, which is why those models’ predictions are given a higher weight than the less accurate models. This ensures the more accurate models contribute more to the predictions, thereby increasing accuracy even more.

Bootstrap Aggregation (Bagging): A resampling technique used to estimate statistics on a population by sampling a dataset with replacement. Similar to K-fold Cross Validation, bagging is used to get a better idea of the performance of a model.

Resampling Ensemble: An ensembling technique that follows the same idea that a collection of similar models are more accurate than each of those models individually. Resampling Ensemble takes a dataset, splits into n\_splits+1 sections, then split each of those into train and test sections, then trains a model on each section and ensembles them together to hopefully have one better model that's more stable than each model together.

K-fold Cross Validation: A model training technique used to better judge the performance of a model than just running it once over the dataset. K-fold Cross Validation takes a dataset, gets k\_folds so that 0.8 of the dataset is for training, 0.2 is for testing. This portion moves throughout the dataset so that by the end, each portion of the dataset got a chance to be the test portion.

# Reviewing of a Training Session

After training many models, the user may want to review the results from previous training sessions.

Main Menu:

---- Pneumothorax Classifier ----

-- Menu --

1) Preprocess scans

2) Train/Test model

3) View Train/Test results

0) Quit

Choice:

The user will choose 3 to review training/testing results

Menu presented:

-- View Results Menu --

Classification type:

1) Binary (Predicting positive or negative)

2) Segmentation (Predicting segments)

0) Quit

Choice:

Because training sessions are categorized by the classification type performed and model architecture, the user must specify those first.

Model architecture:

1) CNN

2) U-net

0) Quit

Choice:

After choosing what classification type and model architecture were relevant to the training session, the user gets to choose the dates containing those types of training sessions.

- Available training session dates -

2019-09-26

2019-09-27

2019-10-01

2019-10-03

2019-10-04

2019-10-15

2019-10-16

2019-10-17

2019-10-22

Date to view (YYYY-MM-DD, -1 to quit):

The user can remember what date they ran a specific training session, or they can just be curious about what training sessions they ran on a specific date. Either way, the user will choose what date to explore. This menu of available training session dates will varied depending on what dates the user ran training sessions.

- Available training session numbers -

1

2

3

Training session number (-1 to quit):

The user can choose what training session number they would like to review for the date they specified. If they ran 5 training sessions on that date, this menu will contain the numbers 1 – 5.

Once the user selects a training session number, they will be displayed the full results of that training session. An example of the output:

cnn\_regular\_model\_1\_of\_1.h5

hyperparameters.json

periphery\_data.json

train\_stats\_1\_of\_1.json

validation\_stats\_1\_of\_1.json

-- hyperparameters corresponding to binary classifier with cnn model architecture

{

"activation\_regularization": 0.0,

"augmented": true,

"bagging\_num\_models": 3,

"balanced": true,

"batch\_normalization": false,

"batch\_size": 10,

"conv\_activation": "relu",

"conv\_layer\_size": 16,

"dataset\_size": 300,

"dense\_activation": "relu",

"dropout": 0.1,

"epochs": 30,

"filter\_size": 3,

"kfold\_cross\_validation\_k\_folds": 5,

"last\_layer\_size": 32,

"loss": "binary\_crossentropy",

"noise\_in\_hidden\_layer": false,

"noise\_std": 0.0,

"num\_conv\_layers": 4,

"optimizer": "adam",

"output\_activation": "sigmoid",

"pool\_size": 3,

"resampling\_ensemble\_n\_splits": 0,

"train\_ratio": 0.7,

"val\_ratio": 0.2,

"weight\_limit": 2,

"weight\_regularization": 0.0001,

"weighted\_avg\_ensemble\_num\_models": 3

}

Saved models:

cnn\_regular\_model\_1\_of\_1.h5

-- Training Results --

#train\_stats\_1\_of\_1

Aggregate stats:

----------------------------------

[ 82, 27]

[ 17, 84]

----------------------------------

Accuracy: 0.7904761904761904

----------------------------------

False Positive Rate: 0.24770642201834858

False Negative Rate: 0.16831683168316836

PPV (Precision): 0.7567567567567568

----------------------------------

Specificity: 0.7522935779816514

Sensitivity: 0.8316831683168316

Total (spec + sens): 1.583976746298483

----------------------------------

ROC: 0.5839767462984831

F1: 0.7924528301886793

MCC: 0.5845079240927633

----------------------------------

-- Validation Results --

#validation\_stats\_1\_of\_1

Aggregate stats:

----------------------------------

[ 20, 6]

[ 10, 24]

----------------------------------

Accuracy: 0.7333333333333333

----------------------------------

False Positive Rate: 0.23076923076923073

False Negative Rate: 0.2941176470588235

PPV (Precision): 0.8

----------------------------------

Specificity: 0.7692307692307693

Sensitivity: 0.7058823529411765

Total (spec + sens): 1.4751131221719458

----------------------------------

ROC: 0.4751131221719458

F1: 0.75

MCC: 0.4708709557974187

----------------------------------

-- Periphery Data --

{}

- Available training session numbers -

1

2

3

Training session number (-1 to quit):

The first part of the output contains the list of files relating to that training session. This includes the trained models (.h5 files), the hyperparameters, training results, validation results, and periphery data. All but the trained models are .json files.

After the list of files, the output contains the hyperparameters of that training session. In this example, the training session was for a binary classifier with CNN model architecture, and the listed hyperparameters were the hyperparameters for that classifier type and model architecture at the time.

After the hyperparameters, the output lists the saved models, specifically their .h5 files.

After the models list, the output lists the training results for each model. It displays the confusion matrix, accuracy, and other statistical analysis metrics. This is in the format is discussed further in [Statistical Analysis](#_Statistical_Analysis) section.

The output lists the same formatted results for the validation dataset.

At the end, the periphery data, if there is any, is displayed in json format as the rest of the data.

# Model Training Future Features

Features to incorporate include mainly model training techniques:

* Transfer Learning
* Grid Search
* Horizontal Voting Ensemble
* Stacked Ensemble

Detailed information available under [Future Improvements](#_Future_Improvements) section

# Step 4: Model Testing

By this step, the user has trained and validated a working model, performing the proper statistical analysis to support high specificity, high sensitivity, low false-positive, low false-negative, and low likelihood of overfitting. The user is ready to test the working model on a test dataset.

# Statistical Analysis

Definitions and equations for the metrics used in calculating the performance of a model.

Confusion matrix:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Model | |
|  |  | 0 | 1 |
| Actual | 0 | TN | FP |
| 1 | FN | TP |

Where TN means True Negative, and TP means True Positive.

Statistical metrics, their descriptions, and their equations.

|  |  |  |
| --- | --- | --- |
| **Metric** | **Description** | **Equation** |
| Accuracy | % of total correct predictions. |  |
| Precision | % of all positive predicted observations that were actually positive. In other words, the percentage of actual positive observations captured as a correct prediction. |  |
| Sensitivity (Recall) | % of correct positive observations predicted. If there are 10 actual positive observations, and the model predicts 8 of them, then it has a sensitivity of 0.8. |  |
| Specificity | % of correct negative observations predicted. Same as sensitivity, but for negative observations. If there are 10 actual negative observations, and the model predicts 8 of them, then it has a specificity of 0.8. |  |
| Negative Predictive Value (NPV) | % of all negative predicted observations that were actually negative. In other words, the percentage of actual negative observations captured as a correct prediction. |  |
| False Positive Rate (FPR) | % of times the model predicts positive observation when it’s actually a negative observation. If there are 10 negative positive observations, and the model predicts 2 as positive, then it has a false positive rate of 0.2. |  |
| False Negative Rate (FNR) | % of times the model predicts negative observation when it’s actually a positive observation. If there are 10 actual positive observations, and the model predicts 2 false negatives, then it has a false negative rate of 0.2. |  |
| ROC Curve Area | ROC curve is a curve created by plotting True Positive Rate vs False Positive Rate. The larger the area under the curve, the better. | ROC curve area = sensitivity – FPR |
| F1 Score | F1 is the harmonic mean of precision and sensitivity. The range is 0-1, with 1 being desirable. |  |
| Mathews Correlation Coefficient (MCC) | A metric designed to measure the quality of binary classifications by taking into account true and false positives and negatives. Regarded as still being a balanced metric even if the classes are very unbalanced. |  |

# Future Improvements

Dataset Future Functionality

Future

* Programmatically split DICOM images into training and test set based on ground truth so that training dataset is 50% positive and 50% negative, while test dataset is a natural representation of the ground truth’s ratio.

Image Preprocessing Future Functionality

Future functionality includes:

* User being able to specify which preprocessing techniques to perform.
* Custom cropping function for radiographs of other body parts.

Model Training Future Features

Features to incorporate include mainly model training techniques:

* Transfer Learning
  + Uses an already well trained model as a starting point for the weights of a new model.
* Grid Search
  + The trial and error of model training to find optimal combination of hyperparameters. Very computationally expensive.
* Horizontal Voting Ensemble
  + The saving of models during a single training session, and then the weighted averaging of them to reduce variance and improve average performance. Useful for very computationally expensive models, so that the user doesn’t have to run multiple expensive training sessions.
* Stacked Ensemble
  + Uses the outputs of a model(s) as input to a new model. Useful for segmentation -> binary classification, as this can be more accurate than regular binary classification.