University of the Philippines Manila

College of Arts and Sciences

Department of Physical Sciences and Mathematics

RaDSS V02: A Radiolarian Classifier Using Convolutional Neural Network

A special problem in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Computer Science

Submitted by:

Micah P. Quisote

May 2018

Permission is given for the following people to have access to this SP:

Available to the general public	Yes
Available only after consultation with author/SP adviser	No
Available only to those bound by confidentiality agreement	No

ACCEPTANCE SHEET

The Special Problem entitled "RaDSS V02: A Radiolarian Classifier Using Convolutional Neural Network" prepared and submitted by Micah P. Quisote in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance.

Geoffrey A. Solano, Ph.D. (*cand.*)
Adviser

EXAMINERS:

	Approved	**Disapproved**
1. Gregorio B. Baes, Ph.D. (*cand.*)		
2. Avegail D. Carpio, M.Sc.		
3. Richard Bryann L. Chua, Ph.D. (*cand.*)		
4. Perlita E. Gasmen, M.Sc. (*cand.*)		
5. Marvin John C. Ignacio, M.Sc. (*cand.*)		
6. Vincent Peter C. Magboo, M.D., M.Sc.		
7. Ma. Sheila A. Magboo, M.Sc.		

Accepted and approved as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Ma. Sheila A. Magboo, M.Sc.
Unit Head
Mathematical and Computing Sciences Unit
Department of Physical Sciences
and Mathematics

Marcelina B. Lirazan, Ph.D.
Chair
Department of Physical Sciences
and Mathematics

Leonardo R. Estacio Jr., Ph.D.
Dean
College of Arts and Sciences

Abstract

Radiolarian assemblages have played a significant role as a biostratigraphic and paleoenvironmental tool used in age-dating, correlation, and studying deep-sea sedimentary rocks that lacks calcareous fossils. The species rapid classification would allow micropaleontologists to proceed further into studying the structure and way of living of these Radiolarians. RaDSS V02 is a deep learning based system that could help researchers in classifying Radiolarian species' microfossil images through image processing and convolutional neural network.

Keywords: Radiolarian, Deep Learning, Convolutional neural networks, Image Recognition, Image processing

Contents

I. Introduction

A. Background of the Study

Radiolarians have long been known for its beauty due to its numerous characteristics. But other than its aesthetic structure, the organism is also very useful and can be used as bio-stratigraphic and paleoenvironmental tool. They are zooplanktons that drift around the oceans and sinks to the bottom of the ocean floor after their death. Because of various reasons such as their existence over 500 million years ago, diversity and abundance, their siliceous skeletons are important on the study of developing history of the life on Earth based on their fossil records. [1]

However, despite of large body of research surrounding the Radiolaria species, the classification of the said organism has proven to be very difficult. Existing traditional methods of identifying radiolarians are based largely on the study of skeletons, observation and cross-referencing with known species. [2]

The use of technology in classifying Radiolaria has not yet given that much attention by some specialists and the concerns of the taxonomical, morphological and paleontological systematics mostly involve creation of information technologies like databases rather than the pursuit of knowledge like classification, species discovery and data creation.

Problems with automatic biological classifications has been explained by some taxonomists, and Ebach et al. believes that we must first resolve certain matters like issues on taxonomy of species before coming up with this kind of technology. Also, there exists some misconception on the combination of technology and taxonomy with the latter being threatened with slow-death. Bik, H.M. states that "taxonomy should be revamped and reborn for the modern age." [3]

Therefore, time consuming conventional methods of biological classification that involves the use of taxonomic keys, consulting reference books, catalogs, collecting, observing, comparing species etc. must improve with automated classification that could make the identification process more rapid to help the micropaleontologist's work done easier.

Machine learning has long been part of image processing and classification systems, and now have made its way through taxonomy. Some researchers focused on the classification of plankton images and the The National Data Science Bowl 2015 competition winner used Deep Neural Network in classifying greyscale images of plankton into one of the 121 classes which achieved a 98% accuracy rate. [4] Research on zooplanktons have also been developed and Dai et al. created ZooplanktoNet, a deep convolutional Network that classified zooplanktons which achieved an accuracy of 93.7%. [5]

Automatic classification system for the Radiolarian species have emerged for the past years. Previous work of Apostol, et al. named RaDSS materialized the Radiolarian classification using Support Vector Machine (SVM) and classified the Radiolarian species into four classes. [6]. SVM is a type of the traditional machine learning where manual feature extraction approach is used, most of the time through an external feature extractor. Apostol et al. used ImageJ, a Java-based image processing and analysis tool to do this. However, manual feature extraction involves algorithms that detects only specific features and may leave behind other useful information that are not covered by these algorithms.

Fimbres-Castro et al. used translation, rotation and scale invariant method to identify the species. [7] Keceley et al. combined hand-crafted and deep features in classifying the organism. [8]

With the advancement of deep learning and its consistent championship on the ImageNet Large Scale Visual Recognition Competition (ILSVRC), the approach has been a trigger behind numerous technologies and innovations. Deep learning is a type of machine learning where manual feature extraction is replaced with layers of algorithms to detect different features. Presently, there are different deep learning architectures that are widely used in different applications, and some of them are Alexnet and Caffenet. AlexNet is one of the pioneers in the ILSVRC to use deep neural networks, while Caffenet is a variation of AlexNet that was created by Caffe developers. Caffe is an open source deep learning framework which is written in C++, with a Python interface. It is easier to understand, implement and switching between CPU and GPU is simply set by a flag.

By using a deep convolutional neural network, RaDSS V02 will become one of these technologies that aims to classify the Radiolarian species from an image. During classification, the micropaleontologist may set a certain threshold value to determine the minimum probability to achieve to classify the radiolarians microfossil image.

B. Statement of the Problem

Classification of the Radiolarian microfossil is normally done by traditional methods of ocular inspection and comparison with published images of radiolarians by micropaleontologists, which is subjective in nature and time consuming. The first version of RaDSS used machine learning employing manual feature extraction which may not contain all the features, subjective since the user selects these from a list, and may skip useful information.

C. Objectives of the Study

The purpose of this study is to develop a tool that automatically classifies microphotographs of radiolarian species name. Specifically, it has the following objectives:

1. The main user is the micropaleontologist that can use the radiolarian classifier model from the developed training module and can:
   (a) Input a radiolarian microfossil image file or directory that contains these images for classification item Apply preprocessing techniques to the radiolarian microfossil image
      i. Resize and stretch the images
      ii. Resize and pad around to maintain aspect ratio
   (b) Set a threshold value that the classification should reach to assign the image into a class. If the maximum probability is less than the threshold value, classify the Radiolarian microfossil image as unknown.
   (c) View the classification results and the probabilities obtained in the result tab
   (d) Download the result file containing the classification of the image, the probabilities of the classificarion and details of the classifier model used via a PDF
   (e) View tutorial/help manual on how to use the radiolarian classifier
2. The AI Expert can use the training module with the following features:
   (a) Input either of the following:
      i. A microfossil image file of a radiolarian species and set its classification

   ii. An directory containing the radiolarian species' microfossil image and an excel file that maps the filenames in this directory to Radiolarian classes or labels

(b) Set the radiolarian species classification of the chosen microfossil image

(c) Apply preprocessing techniques to the radiolarian microfossil image

   i. Resize and stretch the images

   ii. Resize and pad around to maintain aspect ratio

(d) Choose which CNN architecture to train - whether Alexnet or Caffenet

(e) Input the CNN hyperparameters

   i. Input the learning rate - the rate of the neural network on learning

   ii. Input the number of iterations - the maximum number the images will pass through the network

   iii. Input the number of batch size - the number of images to train every iteration

   iv. Input the number of stepsize - the number of iteration to make to drop the learning rate

(f) Build the CNN Model

   i. Train and build the model using Caffe

   ii. Cross validate model to get accuracy

D. Significance of the Project

Radiolarian assemblages can be very useful biostratigraphic and paleoenvironmental tool in dating geological structures. Since classification is difficult in nature, using an automated classifier will greatly help the micropaleontologist in deciding the species class and will help improve this species taxonomy. Deep learning will be useful in analyzing radiolarians complex structure and classify the species using this analysis.

E. Scope and Limitations

1. Species that can be classified by the tool is limited to classes that have at least 20 instances

2. The accuracy of the model is dependent on the number of data that are used on training.

3. Only radiolarian microfossil jpeg images and corresponding excel label files are accepted as input

4. The CNN architectures available for training and classification are fixed.

5. Learning rate, number of iterations, batch size, stepsize are fixed once trained and being used already by the micropaleontologist.

6. Output file generated by the system is available in PDF format

F. Assumptions

1. The system is used by the researcher focusing on Radiolarian species

2. The system serves only as a guide to the researcher. The final classification of the species will be identified by the micropaleontologist.

3. Input excel files for the training contain the labels of the species image. These files are assumed to be correct and nonempty.

II. Review of Related Literature

Microfossils are perhaps the most important group of fossils because they are extremely useful in age-dating, correlation, paleoenvironmental reconstruction especially in the fields of oil industry, mining, engineering and billions of dollars have been made based on microfossil studies. [9]

Vides, an expert system combining artificial intelligence and visual approach is created by Swaby in identifying microfossils. It is in response to the problem of time consuming identification of the microfossil on the deposits of the traditional manual methods. Swaby implemented Lisp and IntelliCorps knowledge engineering environment (KEE) which helped the development of the user interface considerably. A knowledge base builder created a code, data structure, graphical windows complete with image and description to use as reference in identifying the input images. Classification is based on the presence or the absence of certain forms and structure of the species. [10]

ONeill et al. created a system called GeoDAISY which is a modification of the system DAISY (Digital Automated Identification System) to automatically identify microfossils within a commercial stratigraphy environment. Some functionalities were added like a caching mechanism based on Linux memory mapping, pattern correlation and image rotation and scaling. It combined plastic self-organizing map neural network technology and deep learning technique that was able to teach itself partially and achieved 66.67% accuracy. [11]

Wong et al. used a dynamic hierarchical learning algorithm that implements both supervised and unsupervised dynamic learning to accelerate microfossil identification. Digital representations of the specimens are used to form clusters using Agglomerative

Hierarchical Clustering (AHC). Propagation and prioritization was then implemented for the microfossil identification and achieved comparable rates to the best benchmark results obtained using K-NN method. [12]

Deep learning algorithms have dominated the image recognition and classification world. With its rapid development and wide popularity, researchers have implemented the technique on various competitions and tasks to solve real world problems.

Tindall et al. used a convolutional neural network for plankton identification and trained the data on a VGG16 architecture. Planktons are important to marine ecosystem because of their role in the food web. Feeding data on a pre-trained architecture is commonly known as transfer learning. The VGG16 is a 16-layer architecture that consists of convolutional and max pooling layers is pre-trained using the ImageNet dataset. In the paper, they fine-tuned the last layer to learn features of the plankton dataset and achieved 85% accuracy. [13]

Jindal et al. also used CNN as a generic feature extractor with a random-forest classifier on top of the hierarchy. It classified the plankton into 121 types presented by Kaggle and Booz Allen Hamilton. They used various preprocessing and data augmentation techniques before using the input images for training using minibatch stochastic gradient descent with Nesterov momentum that achieved a logarithmic loss of 0.75 which is on the top 10%. [14]

A CNN similar to the VGG architecture was used by Kuang. She implemented the algorithm together with data augmentation, dropout regularization, leaky and parametric ReLu activations and various model assembly methods to achieve classification task. Among the different architectures, the 6 conv + 3 Fully connected layers performs the best on the test set with 0.77 log loss. [15]

The National Data Science Bowl Competition winner also used CNN to classify planktons and coded the program using Python with NumPy and Theano libraries. They performed rescaling and global zero mean unit variance to pre-process the images then augmented the data based on various parameters like rotation, translation, flipping, shearing and stretching. To avoid overfitting, implementation of judicious techniques such as dropout, weight decay, data augmentation, pre-training, pseudo labelling and parameter sharing was necessary. Their convnet architectures consist of lots of conv layers with 3x3 filters and overlapping pooling with window size 3 and stride 2. Leaky ReLu was also implemented instead of only ReLu to introduce nonlinearity to the data. One example of their architecture is composed of 13 layers with parameters (10 conv and 3 fully connected) and 4 pooling layers. Trained using stochastic gradient descent that took 24-28 hours, the best model achieved an accuracy of 82% on the validation set, and a top-5 accuracy of over 98%. [4]

Dai et al., inspired by the AlexNet and VGGNet architectures created ZooplanktoNet, a deep convolutional network to classify zooplanktons automatically and effectively. It aims to capture more general and representative features than previous predefined feature extraction algorithms. Their dataset consists of zooplankton images that involves 13 classes. Rescaling and subtracting the mean value over the training set were done to pre-process the images and data augmentation techniques like rotation, translation, shearing and flipping were also implemented. With a total of 6 conv layers and 3 fully connected layers, the system achieved a 93.7% accuracy [5]

Foraminifera, a species that is also useful in age-dating, was classified using a knowledge based system by Liu et al. To extract descriptive parameters from the given set of images, they used computer vision techniques and the results were then compared against a knowledge based system to infer its class. The knowledge based system consists of descriptions to be used in a rule based classification approach. [16]

Pedraza et al. created an automated classification scheme for diatoms - microfossils that are also studied as paleoenvironmental markers. They were classified into 80 types using a CNN model created after fine tuning a pre-trained AlexNet architecture. It achieved a 99% accuracy. [17]

Some of the implementations of the papers above include various architectures that have been developed by researchers in the deep learning community. The following architectures won the ILSVRC competition starting from 2012, fueled the deep learning movement and have been the foundation of computer scientists in building their own deep model. One architecture that is very popular and the one that started it all is the AlexNet architecture. This deep convolutional neural network needed to classify the 1.2 million images from the ImageNet dataset into 1000 different classes. The network is composed of 5 convolutional layers, some of which are followed by max-pooling layers and dropout layers, and three fully connected layers. ReLu was used to introduce nonlinearity to the data and data augmentation techniques that consisted of image translations, horizontal reflections, and patch extractions. Trained using batch stochastic gradient descent, the model achieved a 15.4% error rate, a first on the ILSVRC competition. [18]

Zeiler and Fergus created ZFNet which is a fine tuning of the AlexNet. This architecture achieved an 11.2% error rate and instead of using an 11x11 sized filters in the first layer, they used a filter size of 7x7. The model used ReLUs for their activation functions, cross-entropy loss for the error function, and was trained using batch stochastic gradient descent.[19]

VGGNet, a 19-layered CNN architecture created by Karen Simonyan and Andrew Zisserman introduced the use of 3x3 filters together with 2x2 max pooling layers. They used ReLu layers after each convolutional layer and trained the model with batch gradient descent. VGGNet achieved a 7.3% error rate. [20]

Szegedy et al. introduced the use of Inception modules in its architecture GoogleNet in which some layers perform in parallel rather than the traditional sequential structure. This 22-layered architecture did not use fully connected layers but used average pool instead and achieved a 6.7% error rate. [21]

Microsoft created its own neural network named ResNet which is a very deep, 152-layered architecture that achieved a 3.6% error rate. They introduced residual blocks, wherein the input is fed on a conv-relu-conv series to obtain a residual mapping which is easier to optimize than the original mapping. This architecture brought about the birth of very deep models and won the 2015 ILSVRC competition. [22]

The previous version of RaDSS that was created by Apostol et al. used Support Vector Machine and principal component analysis to classify the radiolarian species. It is written in Java and extracted the features of the input images using ImageJ and JFeatureLib libraries. It was divided into 2 major functionalities, the training module and the classifier app. The former is to be used by the training administrator who will input some SVM parameters to train the model while the latter will take an input to be classified into 4 classes of the species. [6]

III. Theoretical Framework

A. Radiolarian

Radiolarians are planktonic protists that are among the few groups with comprehensive fossil records available for study. Formally, they belong to the Phyllum Protista, Subphylum Sarcodina, Class Actinopoda, Subclass Radiolaria. [23] They are characterized by their geometric and symmetric structure and live mainly in surface waters with the earliest forms existed during the Cambrian age. Most are somewhat spherical, but a wide variety of shapes exist including cone-like and tetrahedral structures that ranges anywhere from 30 microns to 2 mm in diameter. [2]

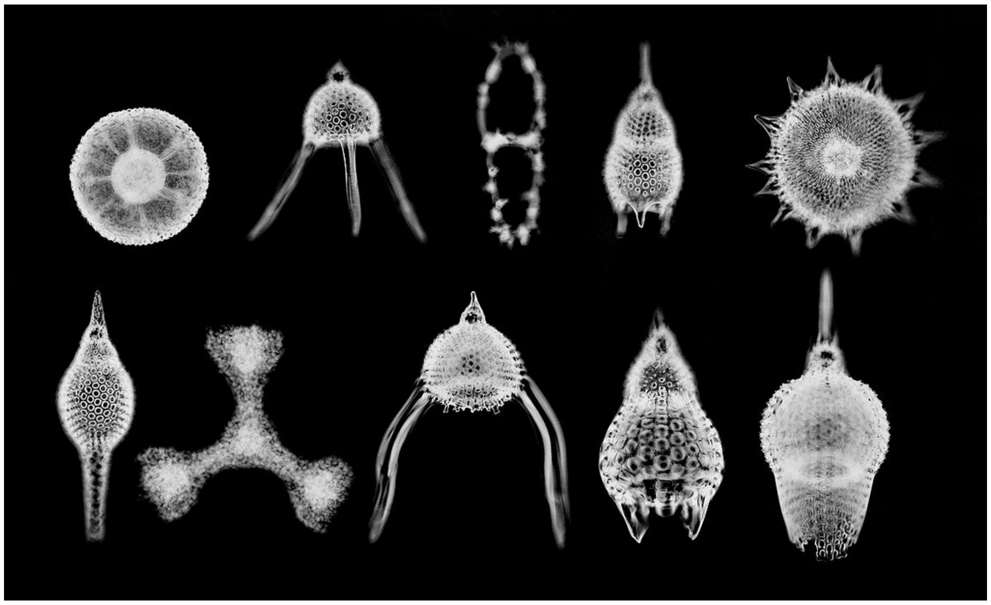


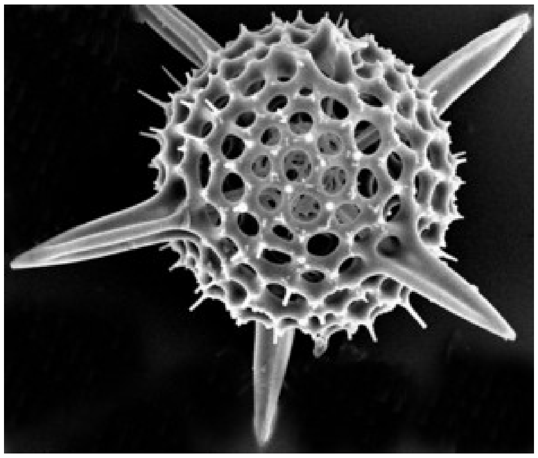
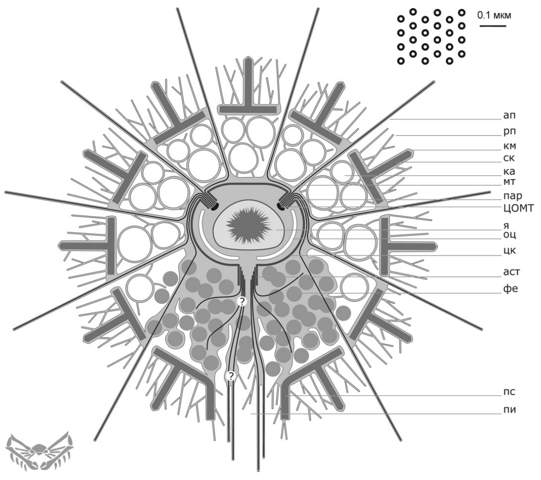
Figure 1. The radiolarian species

This unusual and often strikingly beautiful characteristic of these organisms is their primary morphological characteristic, providing both a basis for their classification and an insight into their ecology. [2]

Radiolarians transition corresponds to three transitions in the geologic time scale namely the Permo-Triassic, Cretaceous-Tertiary and Paleogene-Neogene. They are used in age-dating and the biostratigraphic correlation of oceanic sediments, particularly where calcareous microfossils have been dissolved. [2]

They have been studied extensively by paleontologists because of their well-established presence in the fossil record and unique structure. The current methods for classification are based largely on the study of skeletons from the orders Spumellaria and Nassellaria using features of both the preservable skeleton and the soft parts.

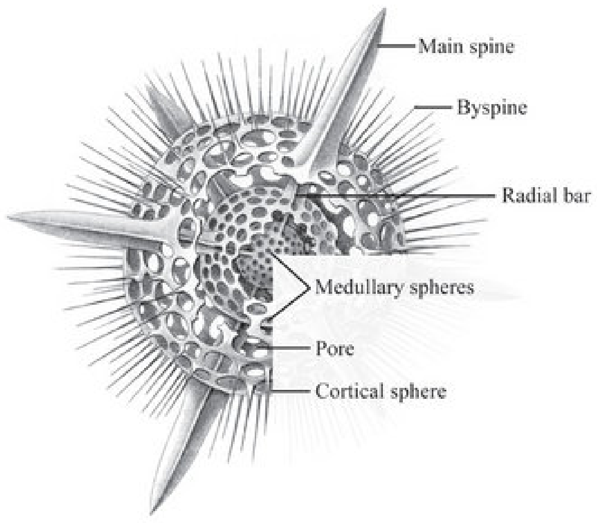
The classification of Radiolaria recognizes two major groups: 1) the Polycystines, with solid skeletal elements of simple opaline silica, and 2) the Phaeodarians, with hollow skeletal elements of a complex siliceous composition that results in rapid dissolution in sea water and consequent rare preservation in sediments.

Polycystines structure and Phaeodarians structure

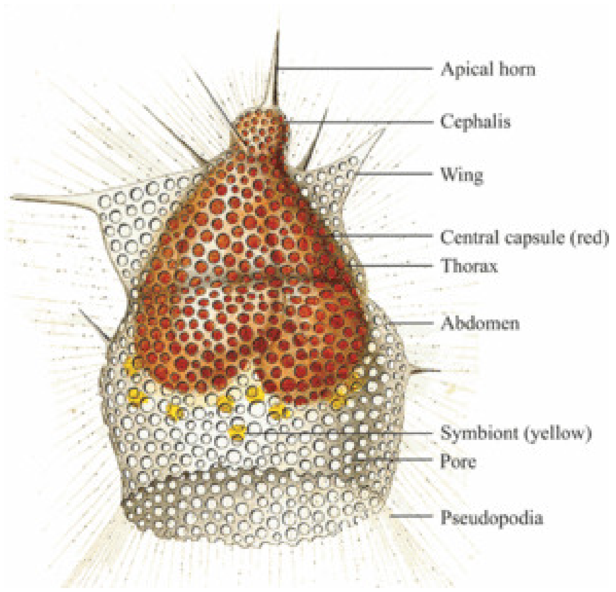
The Phaeodarians also possess a unique anatomical feature, a mass of tiny pigmented particles called the phaeodium. The polycystines, which are the radiolarians best known to geologists, are subdivided into two major groups: the basically spherical-shelled Spumellaria, and the basically conical-shelled Nassellaria. A few polycystine groups lack a skeleton altogether. Some major groups of extinct radiolarians differ substantially from both Spumellaria and Nassellaria, and may be ranked at the same taxonomic level as those groups.

Spumellarians come in various shapes ranging from spherical to ellipsoidal to discoidal. These are the ones typically with radial symmetry. It is common for the Spumellarians to have several concentric shells connected by radial bars. The colonial radiolarians are spumellarians, some with spherical shells and others whose skeletons are instead an association of loose rods, and yet others without skeletons.



Spumellarian structure

Nassellarian shapes derive from geometrically simple spicules (resembling saw horses, "D"-shaped rings, and the like) to which are added (from taxa to taxa) latticed cover to form a chamber, then additional chambers expanding axially into the conical forms typical of the group. [24]



Nassellarian structure

B. Machine Learning

Over time, various researchers have devoted their time and effort in solving real world problems through machine learning. Machine learning is a branch of artificial intelligence that lets the computers learn solving tasks rather than programming them on how to decide on these problems. In order to do this, we take some data, train the model using this data and use its output in predicting relationship or classification based on the given data.

C. Deep Learning

Deep learning or hierarchical learning is a machine learning algorithm that aims to learn data representation or features of the given data as opposed to the previous machine learning algorithms where features are extracted manually. It is composed of layers, commonly stacked together and uses the output of the previous layer as an input to the next layer.

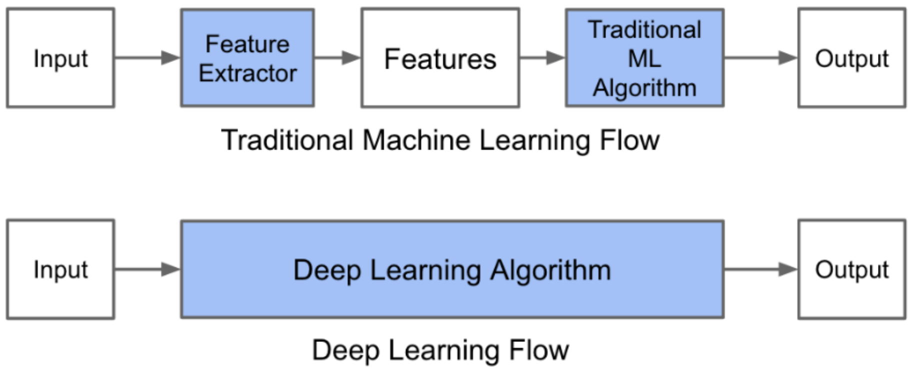


Figure 2. Traditional Machine Learning vs Deep Learning

D. Convolutional Neural Network

Convolutional neural network is the most common deep learning network design for processing data that has known, grid-like topology. [25] It is composed of layers stacked together to extract useful information from the data.

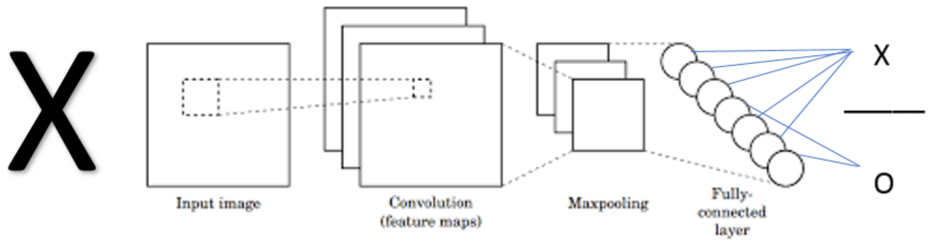


Figure 3. Basic CNN architeure

A typical CNN architecture that makes use of the position of some features in the data to make useful output. The stacking of these layers may vary depending on what order of these layers classifies your data accurately, but the most common stacking includes repeated convolution-activation-pooling layers together, with the fully connected layer as the final layer.

The unique component of CNNs are the convolutional layers. It is composed of convolution an operation on an array of input multiplied to an array of parameters called filter or kernel K resulting in a mapping called feature map. [25]

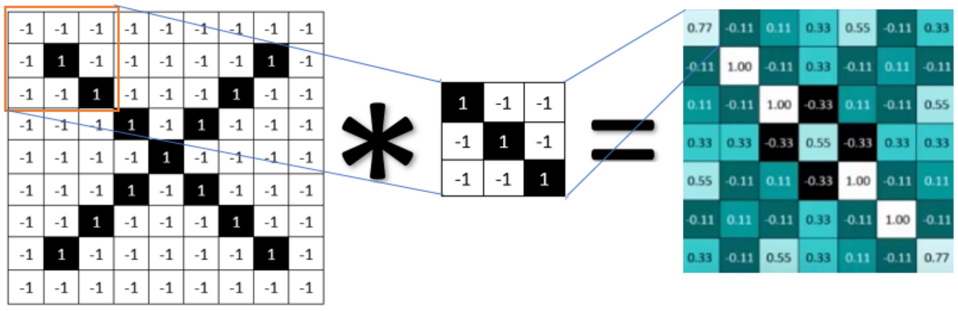


Figure 4. Convolutional layer with 3x3 filter

Above is a convolution operation on an input image represented by pixel values that produced an output array that contains a mapping of a filter to the input. This feature map consists information about the data and will be used as an input to the next layer and so on. High values in the feature map tells us that in this specific position, a diagonal line of that orientation exists. This mapping is only for one feature and in reality, there are other features like edges, lines, curves, patterns, blotches etc. that may be extracted from the data.

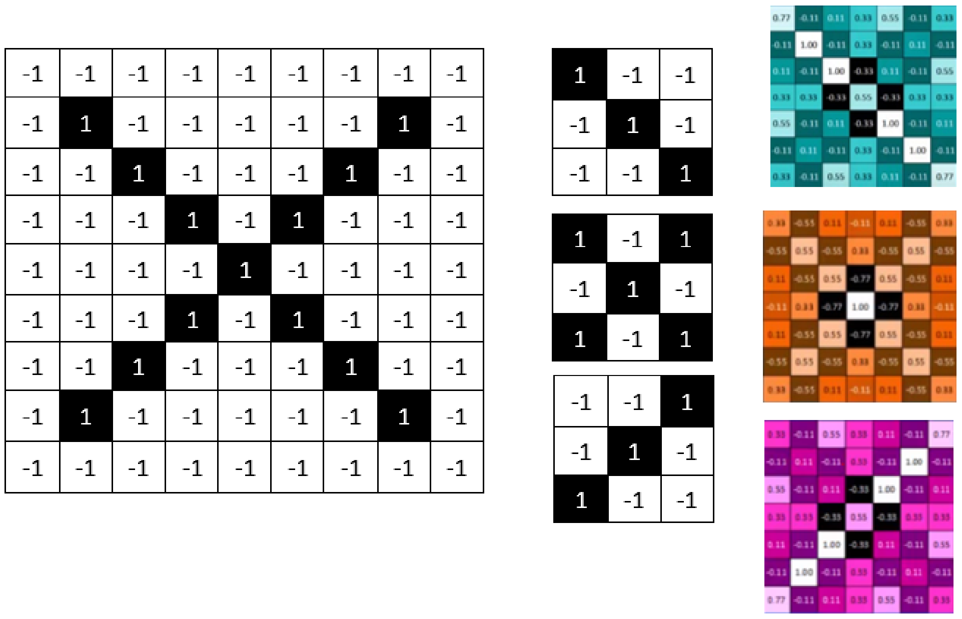


Figure 5. Convolutional layer with three 3x3 filter

The figure above shows three features, two diagonals with different orientation and one x-like feature. All of these are stacked together to form a convolutional layer. After the convolutional layer, an activation function is used to introduce nonlinearity to the data.

Another component of a convolutional neural network is the pooling layer.

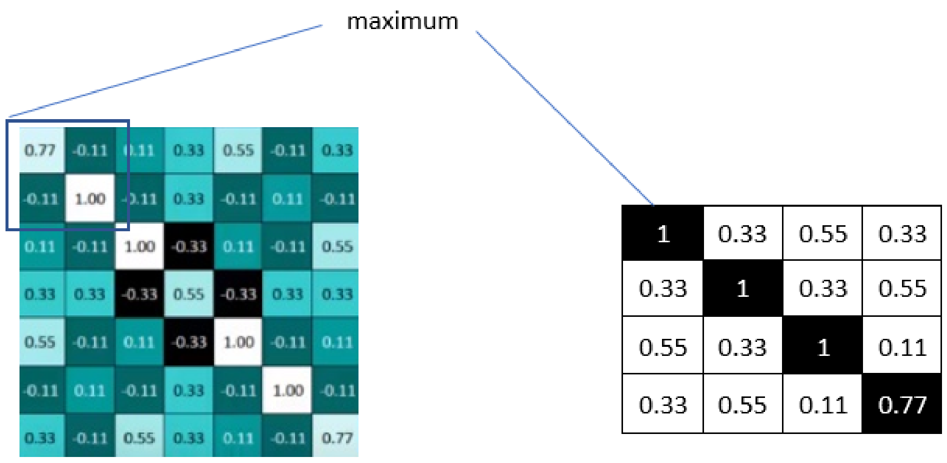


Figure 6. Pooling layer

Pooling methods are performed in the network to reduce the dimensionality of the feature maps thus, lessening computation cost. Pooling helps to make the representation become approximately invariant to small translations of the input, meaning, if we translate the input by a small amount, it doesnt change the values of most of the pooled outputs. [25] It is also used to avoid overfitting, a scenario where the model gets high accuracy in the training set while having a low accuracy in the validation and test sets. Once we know that a specific feature is in the original input volume, i.e. high activation value, its exact location is not as important as its relative location to the other features. [26]

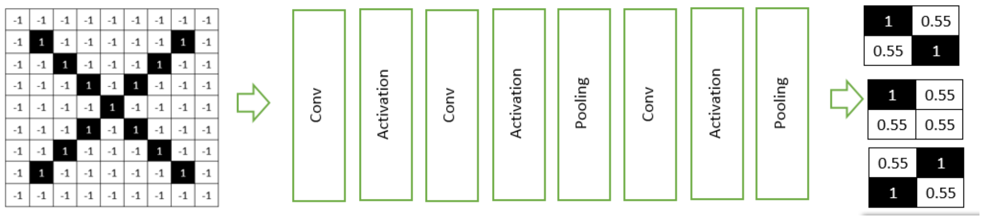


Figure 7. Sample feature map after a series of layers

The figure shows a feature map that have undergone a series of conv-activation-pooling layers reduced into a small dimensional array.

The final layer of a CNN architecture is the fully connected layer which each value votes for a classification. Fully connected layers are usually placed at the end of the network where in every neuron from the previous layer is connected to every neuron in the current layer. This layer will produce an n-dimensional vector where n is the number of classes your model wish to predict. Each number in this n-dimensional vector represents a probability of a certain class. [26]

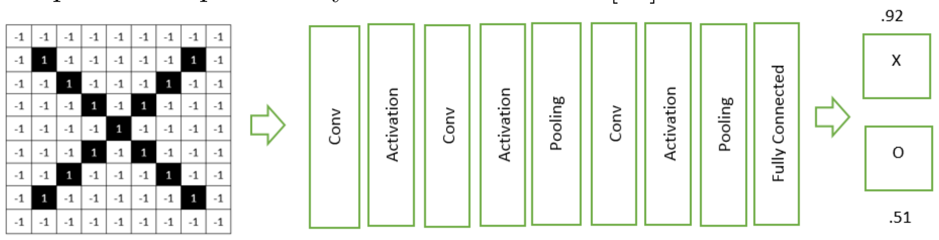


Figure 7. A CNN architecture classifying an input into two classes

D..1 Backpropagation

Convolution and pooling layers act as feature extractors from the input image while fully connected layer acts as a classifier. [27] However, these layers are just the components of the model and its capability to classify will be decided through an iterative training.

Basically, we train our model using the backpropagation algorithm forward pass, loss function, backward pass, and weight update. [27] During the forward pass, the input is fed into the network and outputs a classification. At first iteration, the model will probably classify the image wrongly and the error could be computed using a loss function.

We wanted to have low loss for our model to perform accurately, and to do this, we wanted to know which parameters/weights contributed to that loss through backward pass. We compute the gradient a multi-variable generalization of the derivative, of the error with respect to all weights in the network. After knowing the gradient, we update weights and parameters to minimize the output error.

The process of forward pass, loss function, backward pass, and parameter update is just one training iteration. This process will be repeated for a fixed number of iterations for each set of training images called a batch. Once you finish the parameter update on the last training example, the network should be trained well enough so that the weights of the layers are tuned correctly to create a high classification accuracy. Hyperparameters like the number of iterations, number of epochs, batch size and learning rate will be set before the training process.

IV. Design and Implementation

A. System Design

The system will be implemented using Python. It takes microfossil images of the radiolarian species as input then apply little preprocessing techniques to normalize the data. The pre-processed data is then be fed into the CNN model which will extract the features itself and determine the species classification. The convolutional neural network outputs the predicted species name. The generated outputs can be downloaded at the end by the user in PDF format.
It is divided into 2 major functionalities: the training module and the classifier app. The former builds the Radiolarian classifier model with the specified hyperparameters while the classification of the radiolarian species is done by the classifier app.

A..1 RaDSS Training Module

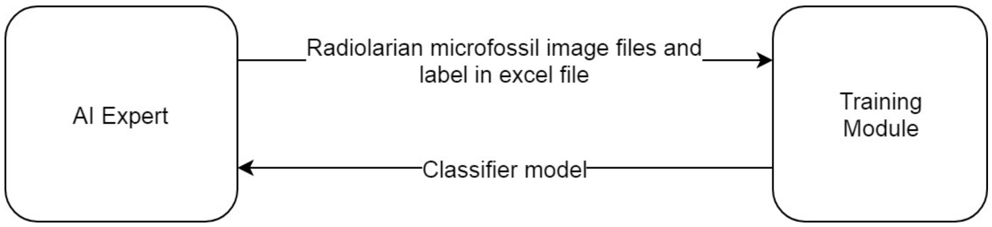


Figure 8. Context Diagram of RaDSS Training Module

The AI Expert builds the classifier model through the training module. A directory and an excel file containing the mapping of the filenames in this directory and their corresponding labels are taken as an input. Furthermore, the user can add additional image to the training and set its clasification once added. The user proceeds to fill up the hyperparameters asked by the application for training. A CNN-based Radiolarian classifier model is generated based on the given Radiolarian images and specified hyperparameters through training. He is given an option to export this classifier model to the classifier application.

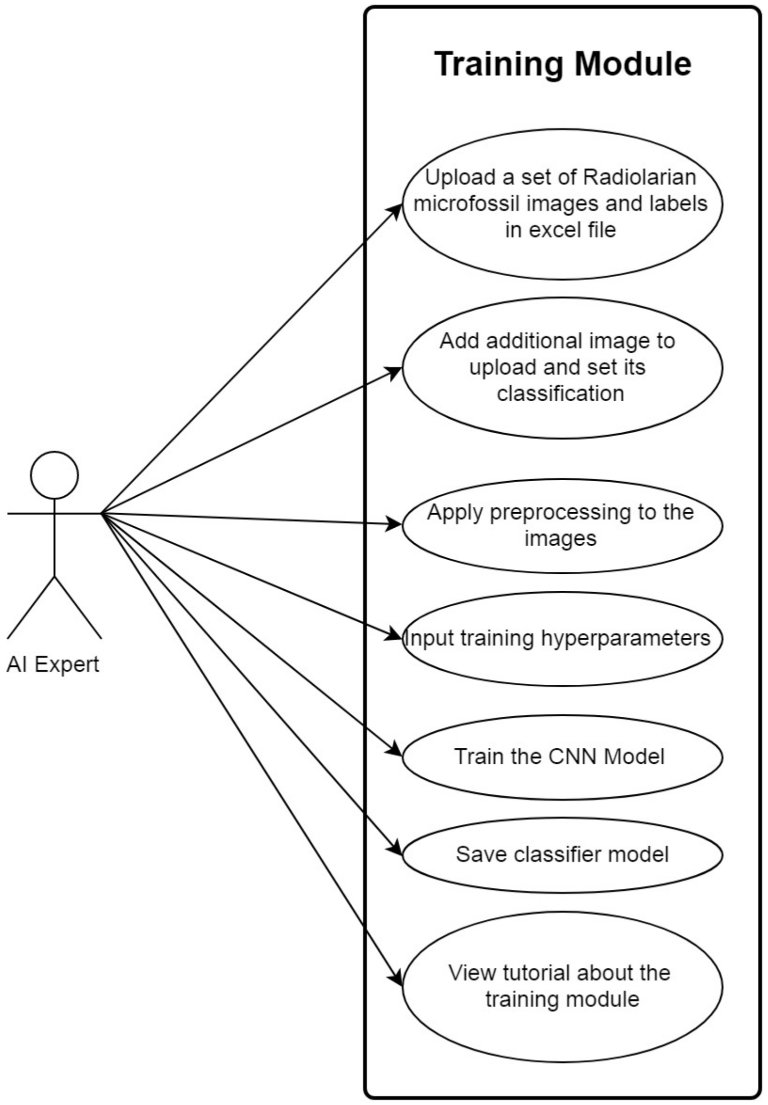

Training Module
Upload a set of Radiolarian microfossil images and labels in excel file
Add additional image to upload and set its classification
Apply preprocessing to the images
Input training hyperparameters
Train the CNN Model
Save classifier model
View tutorial about the training module
AI Expert

Figure 9. Use Case Diagram of RaDSS Training Module

The functionalities that the AI Expert can be used are illustrated in the above use case diagram. The AI Expert can upload image files together with corresponding label file as an input for the training. These images is divided into the training and validation set during training. The former is used to train the model while the latter is used to test the model to obtain accuracy. The user can also edit the classification of the images, input training hyperparameters to train the model including the architecture to use, and finally save the model generated.

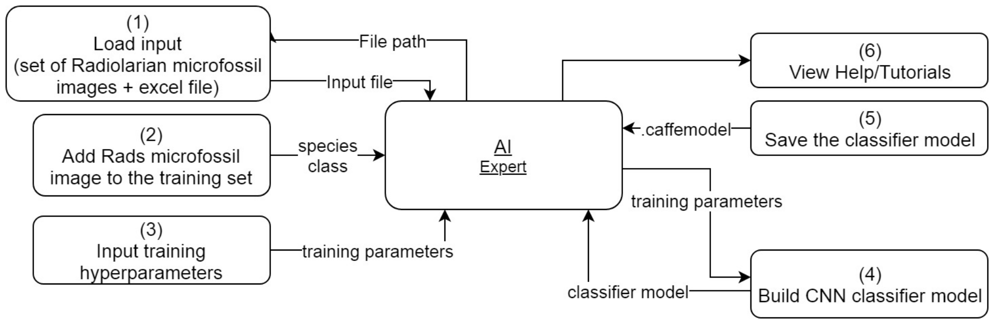

(1)
Load input
(set of Radiolarian microfossil images + excel file)
File path
Input file
(2)
Add Rads microfossil image to the training set
species class
(3)
Input training hyperparameters
training parameters
AI
Expert
(6)
View Help/Tutorials
(5)
Save the classifier model
.caffemodel
training parameters
classifier model
(4)
Build CNN classifier model

Figure 10. Top-level DFD of RaDSS Training Module

The top-level data flow diagram of the training app is shown above. The AI Expert first uploads a directory and an excel file containing the image names with labels or simply just a microfossil image file of the radiolarian species. The user has the option to change the classification of the images.

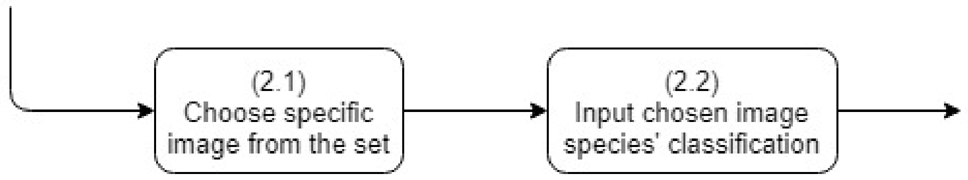

(2.1)
Choose specific image from the set
(2.2)
Input chosen image species' classification

Figure 11. Sub-explosion of process 2 - add species image to training set

Once the training set is completely labeled, the user sets the values of the CNN layers specification and the training hyperparameters to build the model. The AI expert can has the option to export the CNN-based Radiolarian classifier model to the classifier app if this model has a better accuracy. Detailed process regarding the model creation is illustrated below.

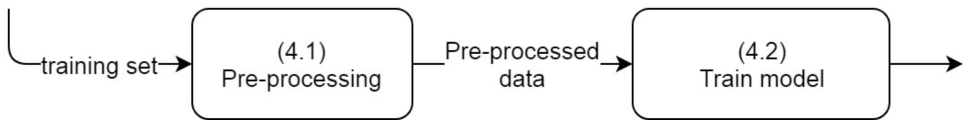


Figure 12. Sub-explosion of process 4 - build CNN classifier model

There are two phases in building the classifier model - the preprocessing and the creation of layers. In the pre-processing phase, there are two techniques involved: resize only and resize and pad. Resize only just resizes the image into 256x256 size while the resize and pad pads around the image before resiizing to maintain aspect ratio.

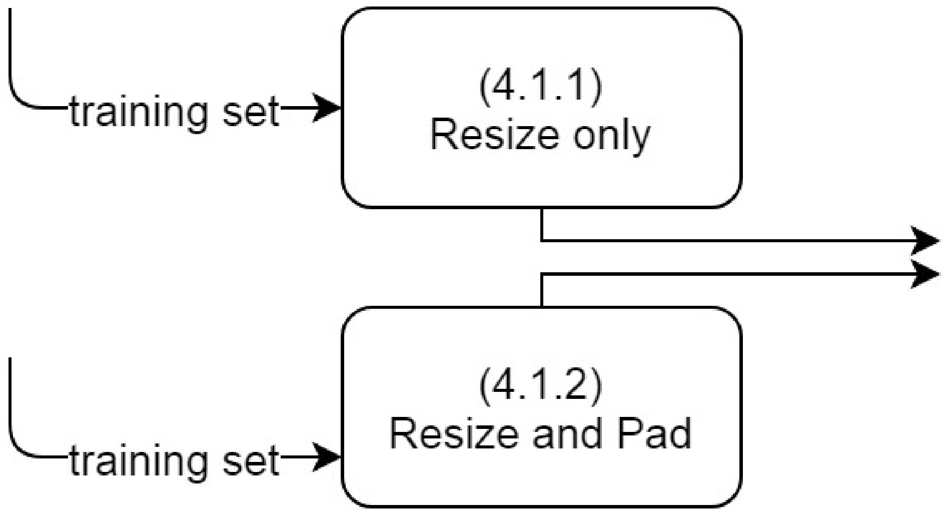


Figure 13. Sub-explosion of process 4.1 - pre-processing

Once the image is pre-processed, we can now train and build the model based on these images.

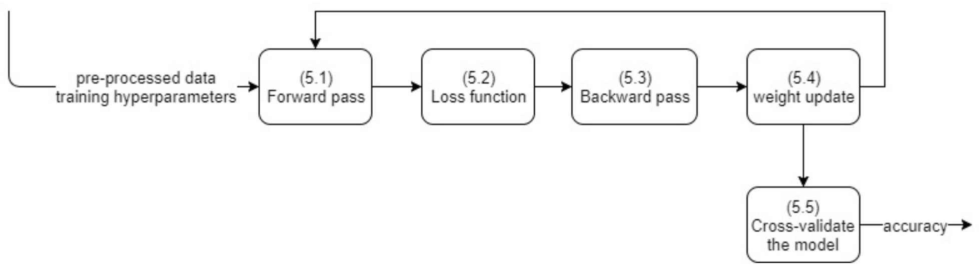


Figure 14. Sub-explosion of process 5 - Train the Model

The training process is comprised of the forward pass, loss function, backward pass and weight update that run until a fixed number of iteration is reached. It ensures that the loss function is low enough to provide an accurate classification. After training, it is then cross-validated using the validation set to know its accuracy.

The outcome of the whole training app is the Radiolarian classifier model. If the AI expert is satisfied with the built model, he has the option to export this classifier model to the classifier application.

The system provides documentation to teach the user on how to use the training module.

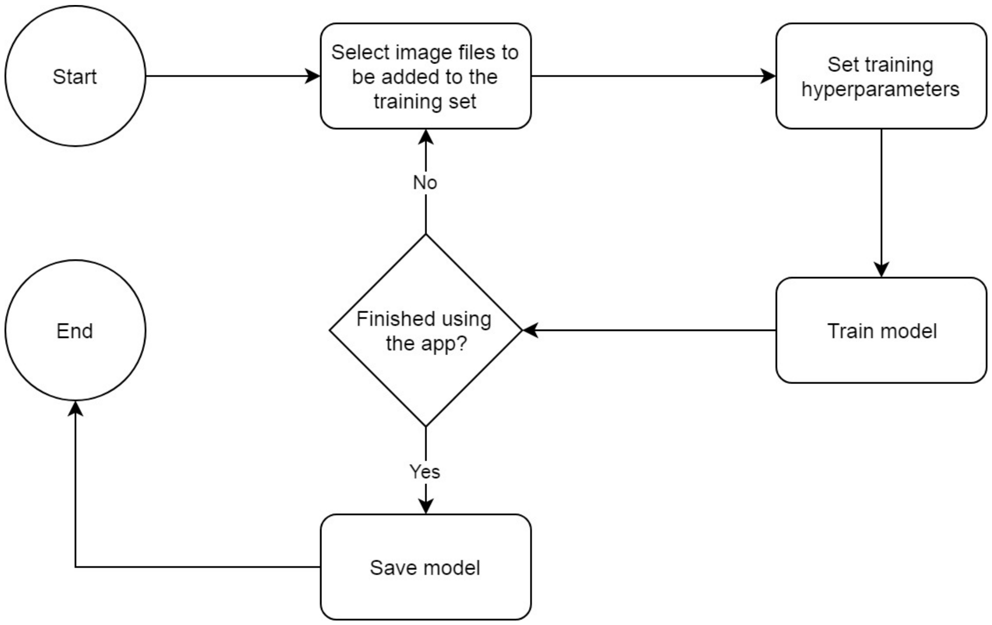

Start
Select image files to be added to the training set
Set training hyperparameters
No
Finished using the app?
End
Train model
Yes
Save model

Figure 15. Activity Diagram of Training Administrator, RaDSS v02

A..2 RaDSS Classifier App

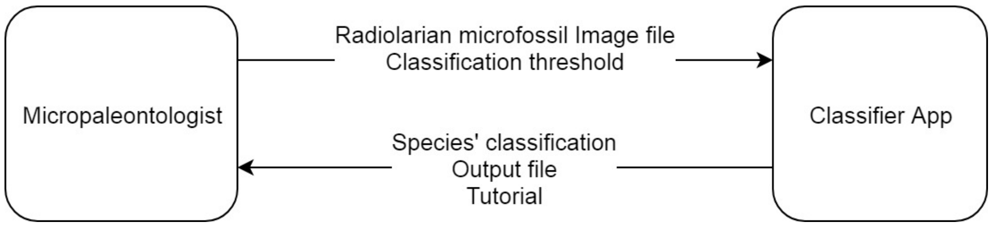


Figure 16. Context diagram of RaDSS Classifier App

The classifier app is a desktop application that can be run offline wherein the micropaleontologist uploads a Radiolarian microfossil image to predict its possible classification. The user can also adjust the classification threshold that is used for the classification process. The use case diagram is illustrated in the image below.

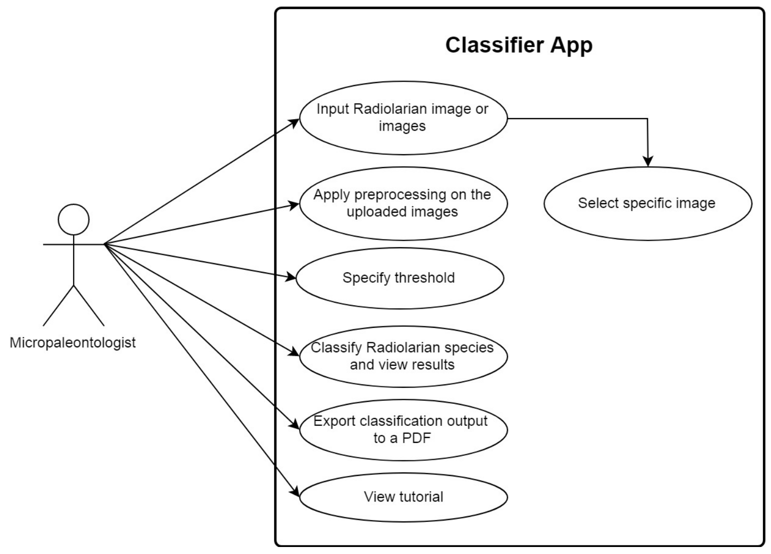

Classifier App
Input Radiolarian image or images
Select specific image
Apply preprocessing on the uploaded images
Specify threshold
Classify Radiolarian species and view results
Export classification output to a PDF
View tutorial
Micropaleontologist

Figure 17. Use case diagram of RaDSS Classifier App

As shown in the use case diagram, the functionalities that the micropaleontologists can use are: upload image files as input, adjust the classification threshold, classify he unknown species using the classifier model trained on the training module, then download the results. He can also read a tutorial on how to use the application.

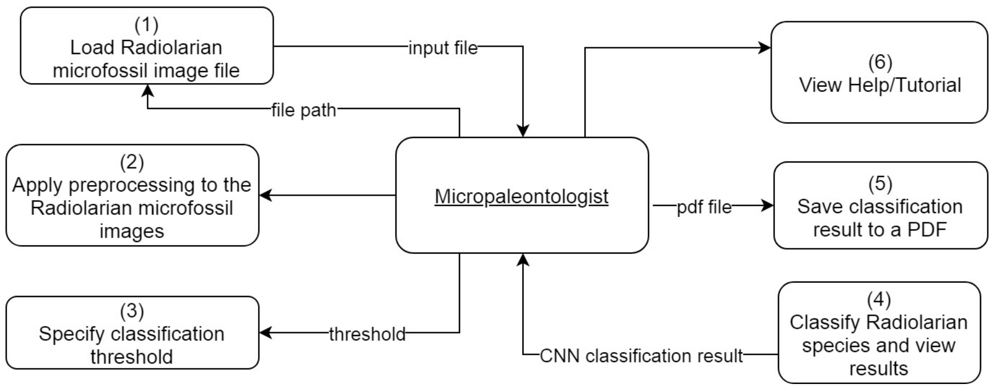

(1) Load Radiolarian microfossil image file
input file
file path
(6) View Help/Tutorial
(2) Apply preprocessing to the Radiolarian microfossil images
Micropaleontologist
pdf file
(5) Save classification result to a PDF
(3) Specify classification threshold
threshold
CNN classification result
(4) Classify Radiolarian species and view results

Figure 18. Top-level DFD of RaDSS Classifier App

The figure above shows the top-level data flow diagram of the system. The micropaleontologist inputs an unknown Radiolarian microfossil image to be classified to one of the classes provided by the system. The image is then pre-processed then fed into the Radiolarian classifier model for classification. Further details can be seen in the following diagram.

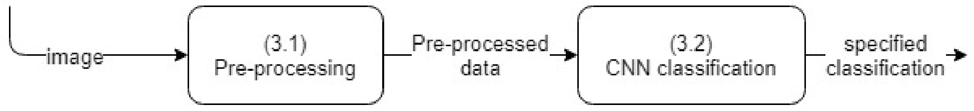


Figure 19. Sub-explosion of process 3 - classify species

Pre-processing will undergo the same methods employed in the training set in Figure 14. After the image is pre-processed, it is fed into the Radiolarian classifier model for classification.

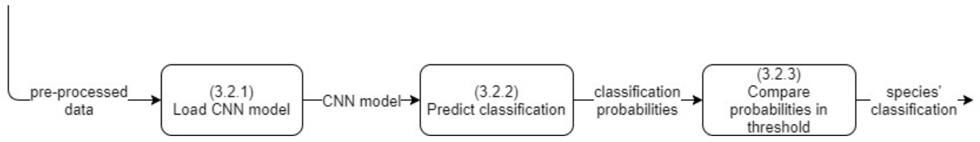


Figure 20. Sub-explosion of process 3.2 - CNN classification

The Radiolarian classifier model trained in the training module that is loaded into the classifier app to predict the classification of the unknown species. Result of the classification includes the possible name with its corresponding probability and then compared to the threshold value. The unknown species is classified as the class with the highest probability greater than the defined threshold. The species is classified as UNKNOWN: If all of the probabilities are less than the threshold.

After generating all the results, the user can then save it in a PDF file including the image, the classification, the probability, and the details of the classifier model use.

An option to adjust the classification threshold is also available for the micropaleontologist and is used to determine the predicted class of the unknown species.

Finally, the user can view tutorials and read the documentation on how to use the system. Screenshots and help instructions is provided together with the summary of commands, panes and operations offered by the application.

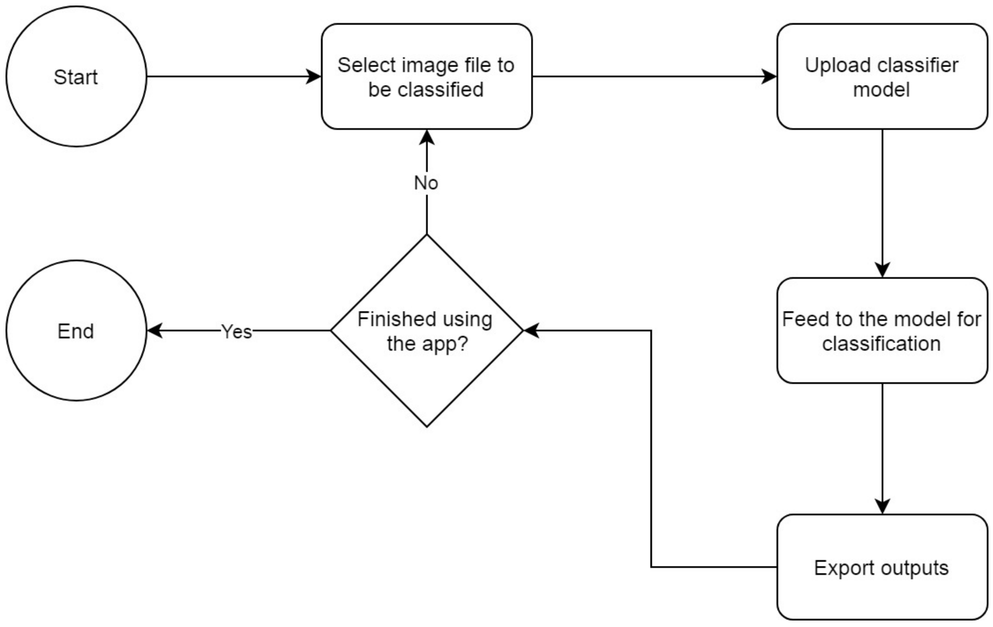


Figure 21. Activity Diagram of Micropaleontologist , RaDSS v02

B. System Architecture

The proposed system is implemented using Python. It is the language that is most widely by data scientists in the deep learning field. Python, together with the Caffe framework is used to train the model.

Caffe is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. It supports CNN and other deep learning network designs. The models are defined by configuration without hard-coding and training can switch between CPU and GPU by setting a single flag. [28]

C. Technical Architecture

1. Windows 7 or Higher;
2. 2 Ghz CPU
3. RAM: 8 GB or higher
4. Graphical Processing Unit (GPU), preferably NVIDIA
5. 2 Gigabytes disk space

V. Results

The Radiolarian Decision Support System v02 is a standalone desktop application created using python. It is divided into two parts, the Training Module and the Classifier Application that can be used by the AI expert and the microplaeontologist respectively.

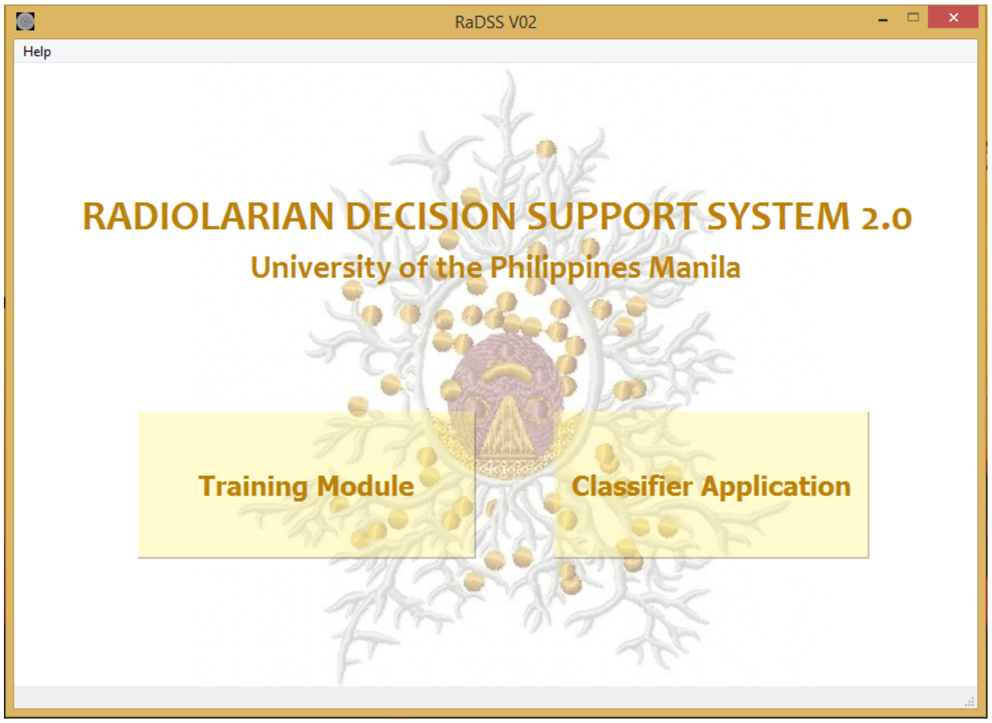
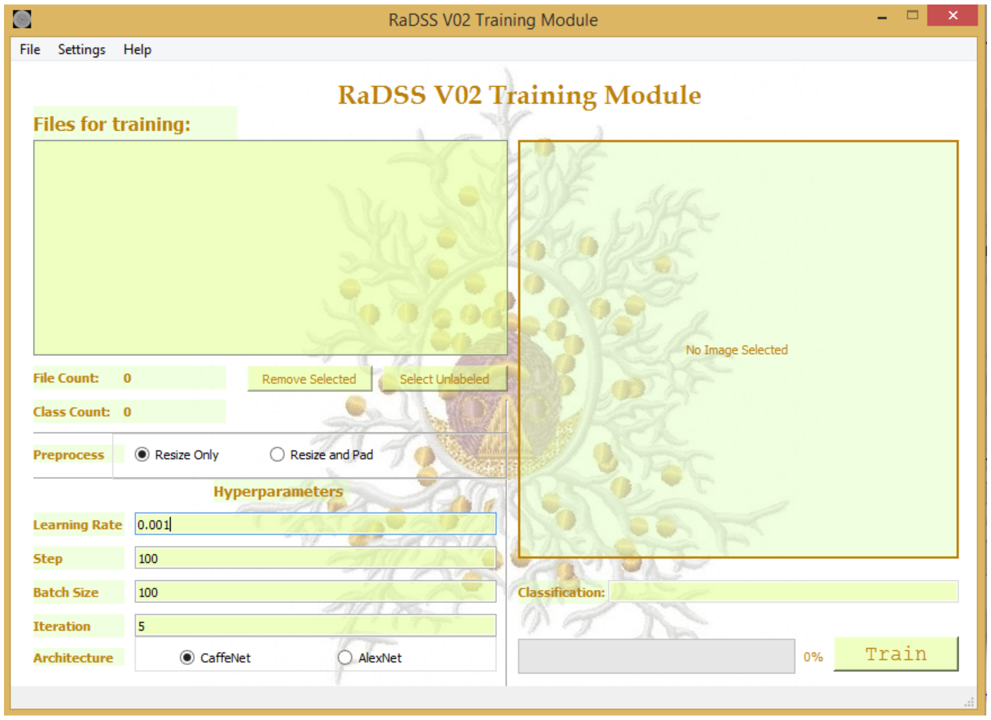

RaDSS V02
Help
RADIOLARIAN DECISION SUPPORT SYSTEM 2.0
University of the Philippines Manila
Training Module
Classifier Application

Figure 20. RaDSS v02 Main Window

A. RaDSS Training Module



File Count: 0
Class Count: 0
Preprocess: (● Resize Only) ○ Resize and Pad
Learning Rate: 0.001
Step: 100
Batch Size: 100
Iteration: 5
Architecture: (● CaffeNet) ○ AlexNet

Figure 21. RaDSS v02 Training Module Startup

To start, the training administrator or the AI expert can add images or an image to the training application. Adding images requires an additional excel file that contains the mapping of the filenames and their corresponding labels. When adding an only image the class label will be specified at the bottom of the preview image.

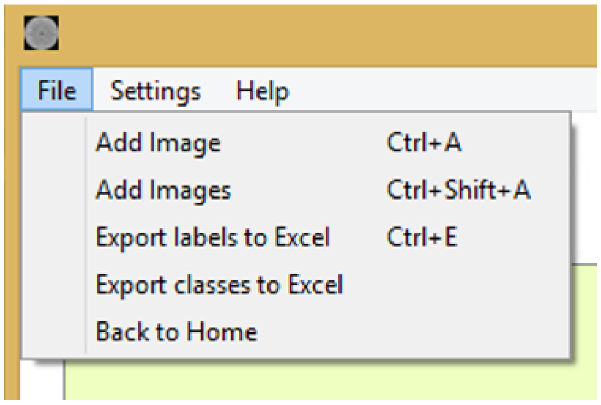


Figure 22. RaDSS v02 Training Module File Options

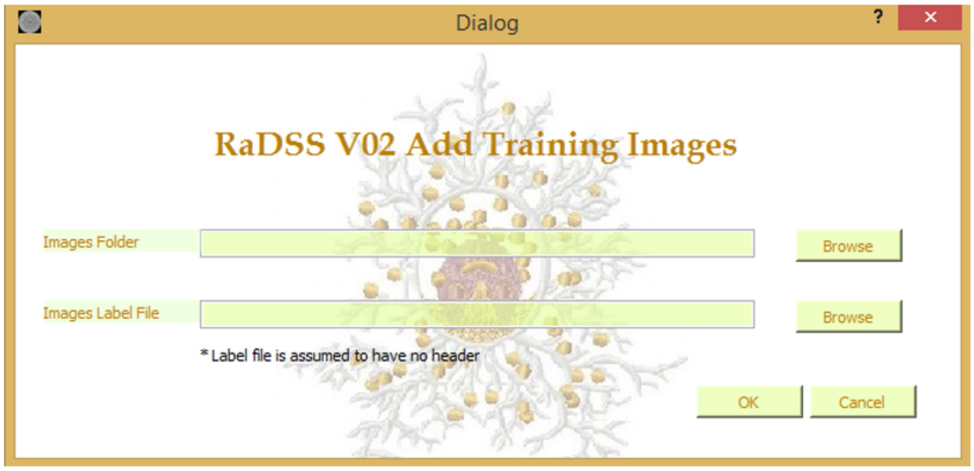
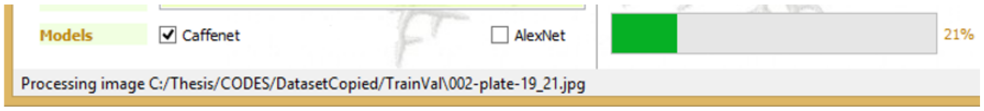

Dialog
RaDSS V02 Add Training Images
Images Folder
Browse
Images Label File
Browse
* Label file is assumed to have no header
OK
Cancel

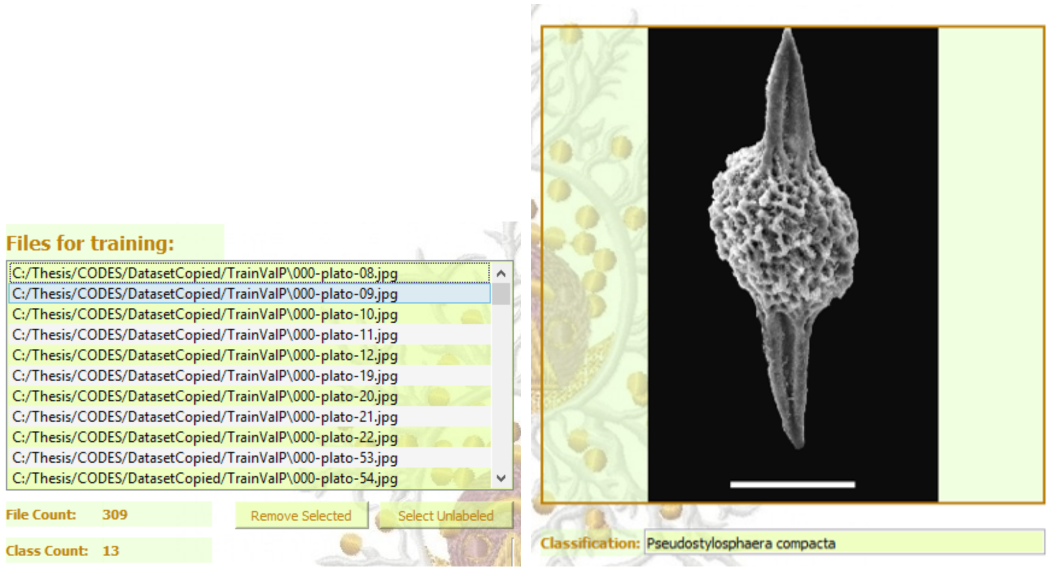
Figure 23. RaDSS v02 Training Module Adding Images

You can see the status of the upoad on the staus bar below and the progress bar beneath the Train button.



Processing image C:/Thesis/CODES/DatasetCopied/TrainVal\002-plate-19_21.jpg

When the images are finally added, it will be listed on the left panel. Cliicking an image will display a preview and its corresponding classification based on the uploaded excel file on the right. He can edit the classification of the selected image but it will not edit the excel file that was uploaded. He can export these updated labels to an excel file by clicking on the "File" menu and choose "Export labels to Excel."



```
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-08.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-09.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-10.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-11.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-12.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-19.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-20.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-21.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-22.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-53.jpg
C:/Thesis/CODES/DatasetCopied/TrainValP\000-plato-54.jpg
```

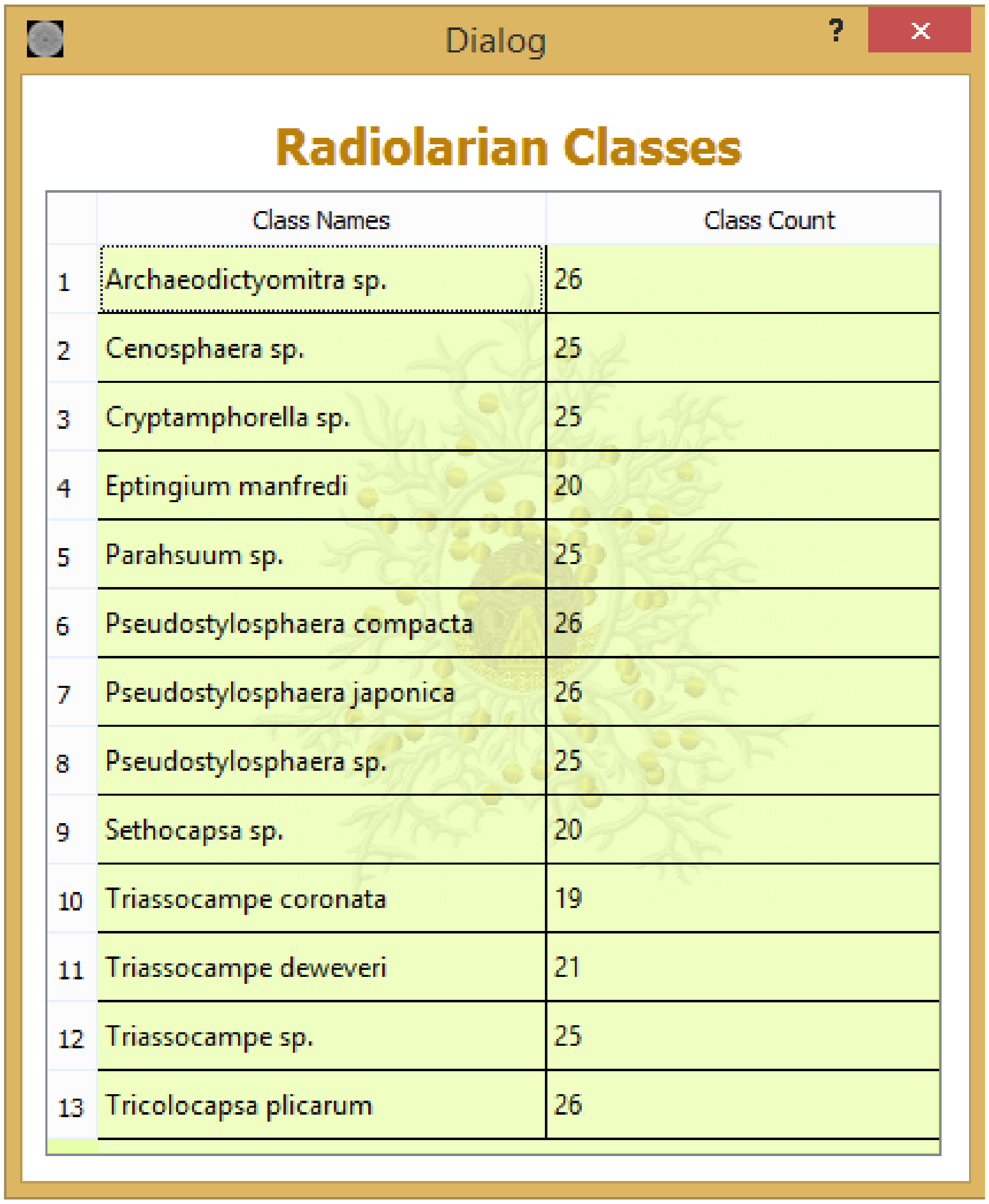
File Count: 309

Class Count: 13

Classification: Pseudostylosphaera compacta

Figure 25. RaDSS v02 Training Module Image List on the right; image preview and classification on the left

Class Count is the total number of classes on the list. You can view these classes and corresponding counts by clicking Help then Show Classes on the menu bar. The user can export the Radiolarian classes and their corresponding class count into an excel file by clicking "File" menu and select "Export classes to Excel"


Dialog

Radiolarian Classes

	Class Names	Class Count
1	Archaeodictyomitra sp.	26
2	Cenosphaera sp.	25
3	Cryptamphorella sp.	25
4	Eptingium manfredi	20
5	Parahsuum sp.	25
6	Pseudostylosphaera compacta	26
7	Pseudostylosphaera japonica	26
8	Pseudostylosphaera sp.	25
9	Sethocapsa sp.	20
10	Triassocampe coronata	19
11	Triassocampe deweveri	21
12	Triassocampe sp.	25
13	Tricolocapsa plicarum	26

Figure 27. RaDSS v02 Training Module Class Count Summary

The preprocess option is also available. The user gets to choose whether the images are going to be resized only or be pad around to maintain aspect ratio, before training.

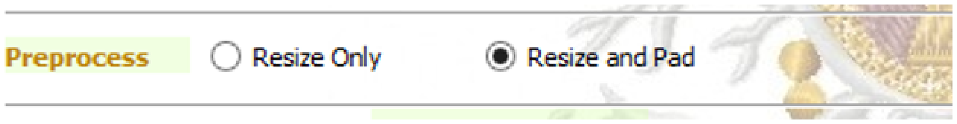
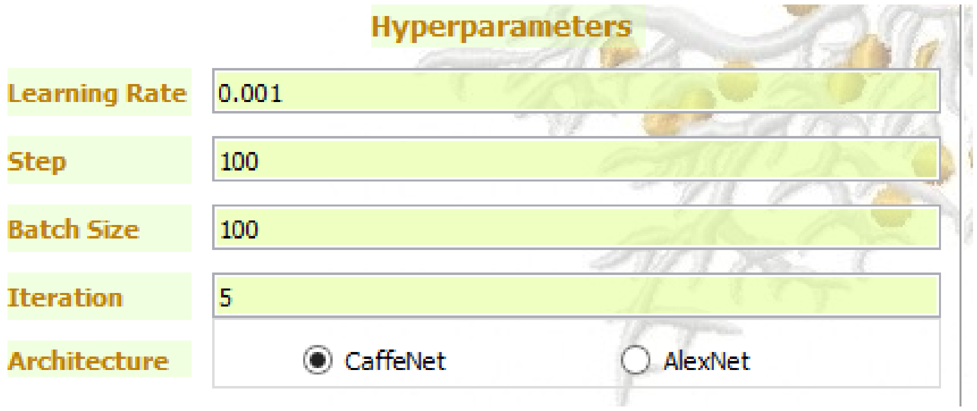

Preprocess ○ Resize Only [✓ Resize and Pad]

Figure 28. RaDSS v02 Training Module Preprocessing

To train a Radiolarian classifier model, the training administrator will input the hyperparameters.



Hyperparameters

Learning Rate	0.001
Step	100
Batch Size	100
Iteration	5
Architecture	[✓ CaffeNet] AlexNet

Figure 29. RaDSS v02 Training Module Hyperparameters

The hyperparameters are the most important part of the module. These are the specifications to be used during training. **Learning Rate**: The default is set to 0.001 and it should be small as possible. This will set how fast or slow the network will learn. Too high learning rate may result to overfitting while too low value may result to an underfit.

Step: Step or stepsize drops the learning rate every step iterations. Meaning, the learning rate will drop on the specified step iteration.

Batch Size: Batch size is the number of images that will be used to train the network per iteration. It is usually a divisible of the total number of training images.

Iteration: Iteration is the total number of passes the images to the network.

Models: Models are the currently available architectures or neural nets that will be trained using the training images. These architectures have once been the state-of-the-art on the image classification fields.

The training is displayed on an external command prompt that can be viewed an monitor by the AI expert. The trained model will be saved on the models path that can be edited in the configuration. It is saved in a .caffemodel format and automatically be made available on the classifier application.

After training, the resulting plot and the training accuracy is displayed. This accuracy is computed based on the model's performance on the validation set. The AI expert has the option to export this caffemodel to the classifier application if it has better performance than that on the classifier application.

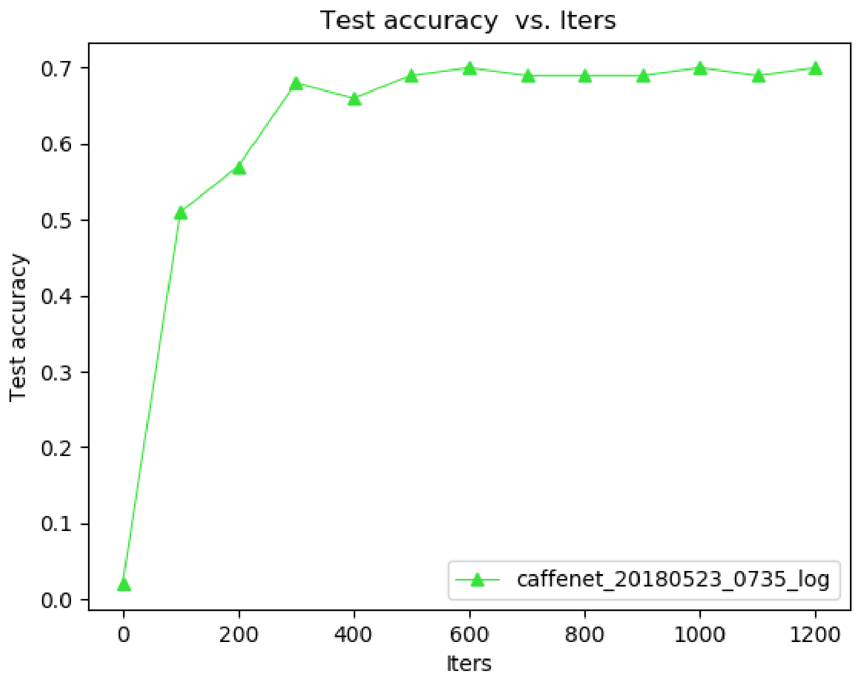
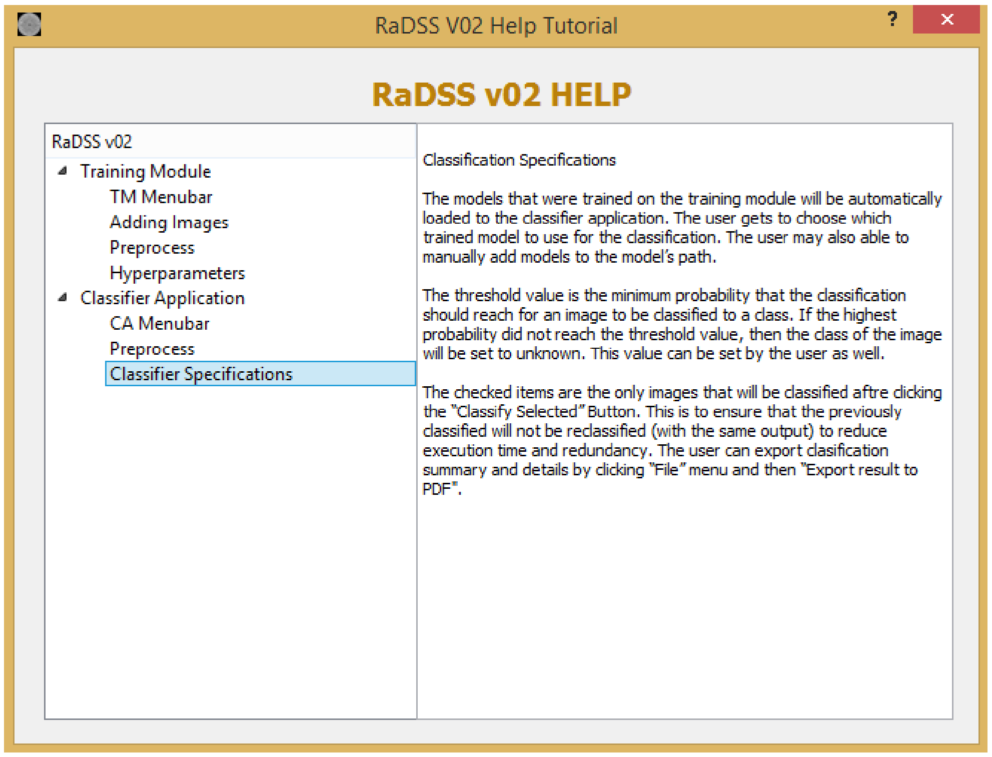


Figure 30. RaDSS v02 Training Module Plot of Training

More information about the different modules and how to use the application can be found in the User's Manual and can be viewed by clicking "Help/Tutorials" under "Help" menu.



Classification Specifications

The models that were trained on the training module will be automatically loaded to the classifier application. The user gets to choose which trained model to use for the classification. The user may also able to manually add models to the model's path.

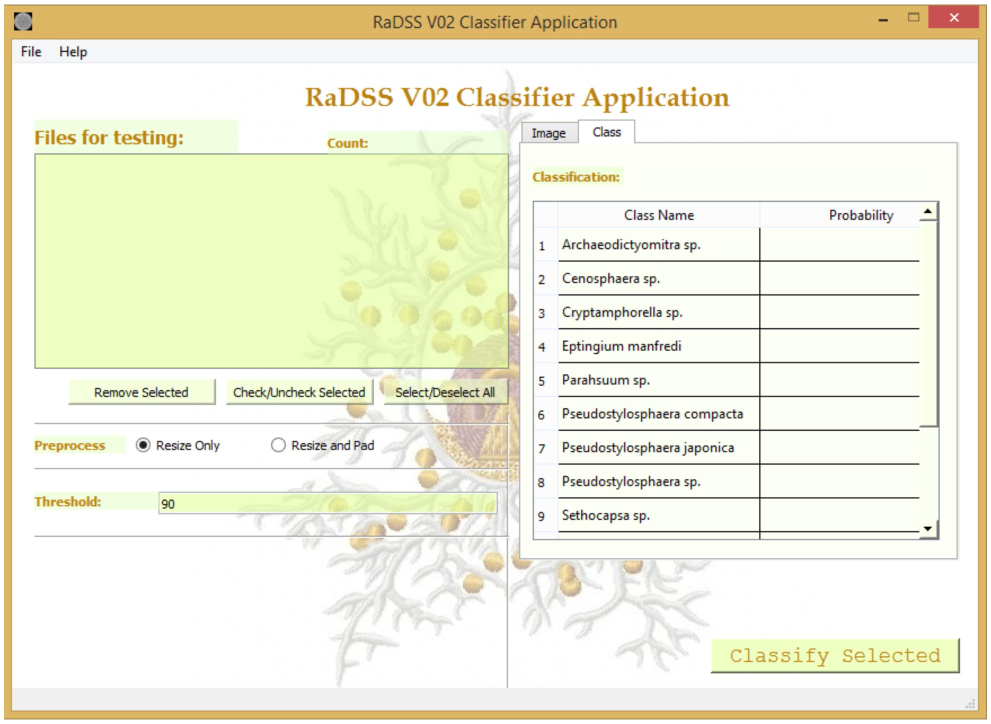
The threshold value is the minimum probability that the classification should reach for an image to be classified to a class. If the highest probability did not reach the threshold value, then the class of the image will be set to unknown. This value can be set by the user as well.

The checked items are the only images that will be classified aftre clicking the "Classify Selected" Button. This is to ensure that the previously classified will not be reclassified (with the same output) to reduce execution time and redundancy. The user can export clasification summary and details by clicking "File" menu and then "Export result to PDF".

Figure 31. RaDSS v02 Training Module Tutorials

The Radiolarian classifier models created in this module plays a significant role in the classifier application. It will be used to predict the possible classification of the unknown Radiolarian species.

B. Classifier Application



Preprocess: [✓ Resize Only] Resize and Pad

Threshold: 90

Classification:

	Class Name	Probability
1	Archaeodictyomitra sp.	
2	Cenosphaera sp.	
3	Cryptamphorella sp.	
4	Eptingium manfredi	
5	Parahsuum sp.	
6	Pseudostylosphaera compacta	
7	Pseudostylosphaera japonica	
8	Pseudostylosphaera sp.	
9	Sethocapsa sp.	

Figure 32. RaDSS v02 Classifier Application Startup

To start using the application, the micropaleontologist can either input a directory of images or an image for classification. Only jpeg images are accepted by the system.

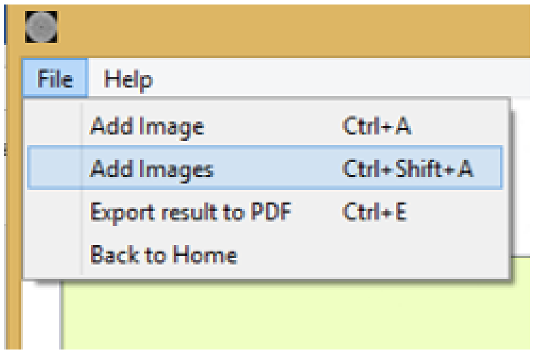
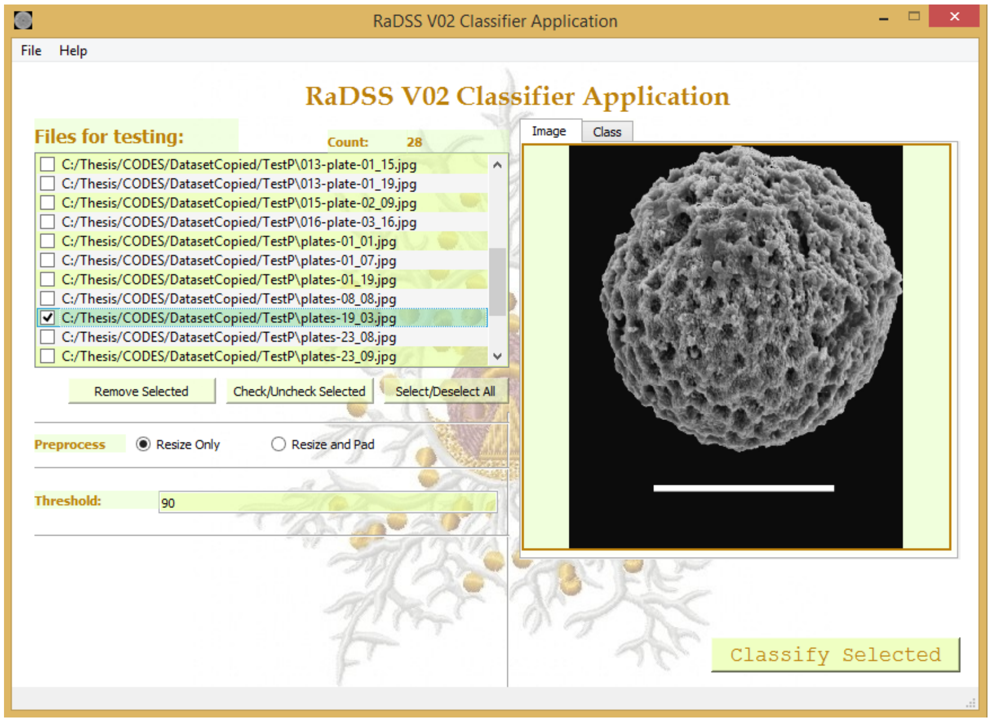


Figure 33. RaDSS v02 Classifier Application File Options

After choosing a directory, the image files will be loaded to the list on the left, showing the file count on the upper right of the list. The user can also view the image preview on the right tabbed pane. The 1st tab contains image, while the 2nd tab provides the summary of the classification, i.e. the class name and the probabilities obtained per class after classification. The 2nd tab values are empty while Classify Selected button have never been clicked.



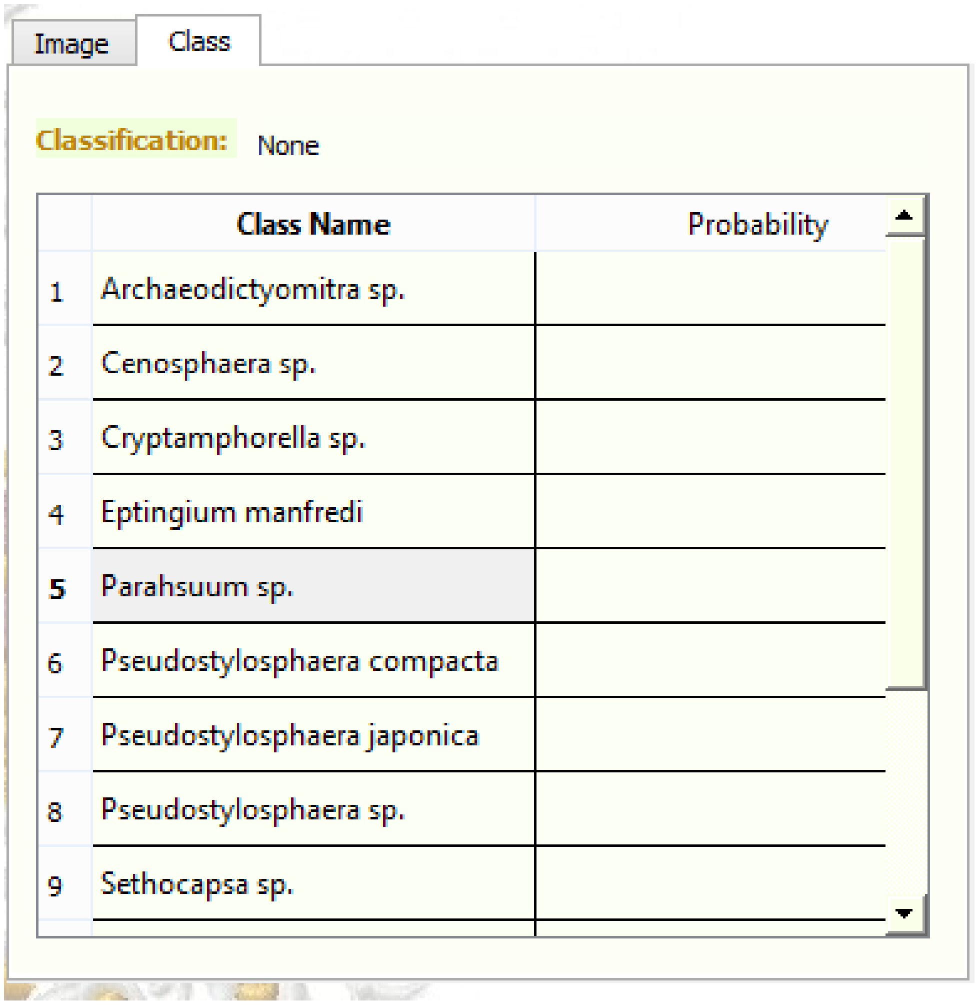
Files for testing: Count: 28

- [ ] C:/Thesis/CODES/DatasetCopied/TestP\013-plate-01_15.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\013-plate-01_19.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\015-plate-02_09.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\016-plate-03_16.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_01.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_07.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_19.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-08_08.jpg
- [✓] C:/Thesis/CODES/DatasetCopied/TestP\plates-19_03.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-23_08.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-23_09.jpg

Preprocess [✓ Resize Only] Resize and Pad

Threshold: 90

Figure 34. RaDSS v02 Classifier Application Uploaded Images

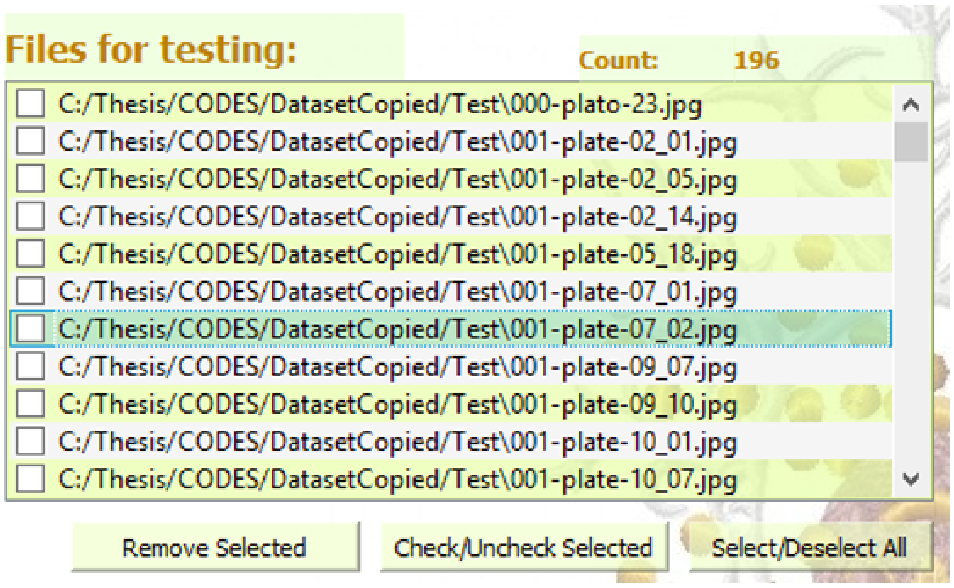


Classification: None

	Class Name	Probability
1	Archaeodictyomitra sp.	
2	Cenosphaera sp.	
3	Cryptamphorella sp.	
4	Eptingium manfredi	
5	Parahsuum sp.	
6	Pseudostylosphaera compacta	
7	Pseudostylosphaera japonica	
8	Pseudostylosphaera sp.	
9	Sethocapsa sp.	

Figure 35. RaDSS v02 Classifier Application Class Tab

Remove selected button will remove the selected files on the list. Select/Deselect All button will select or deselect all images in the list. The Check/Uncheck Selected will check or uncheck all of the selected images. The checked items are the only images that will be classified aftre clicking the Classify Selected Button.



Files for testing:

Count: 196

- [ ] C:/Thesis/CODES/DatasetCopied/Test\000-plato-23.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-02_01.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-02_05.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-02_14.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-05_18.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-07_01.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-07_02.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-09_07.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-09_10.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-10_01.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/Test\001-plate-10_07.jpg

Figure 36. RaDSS v02 Classifier Application List of Images and Options

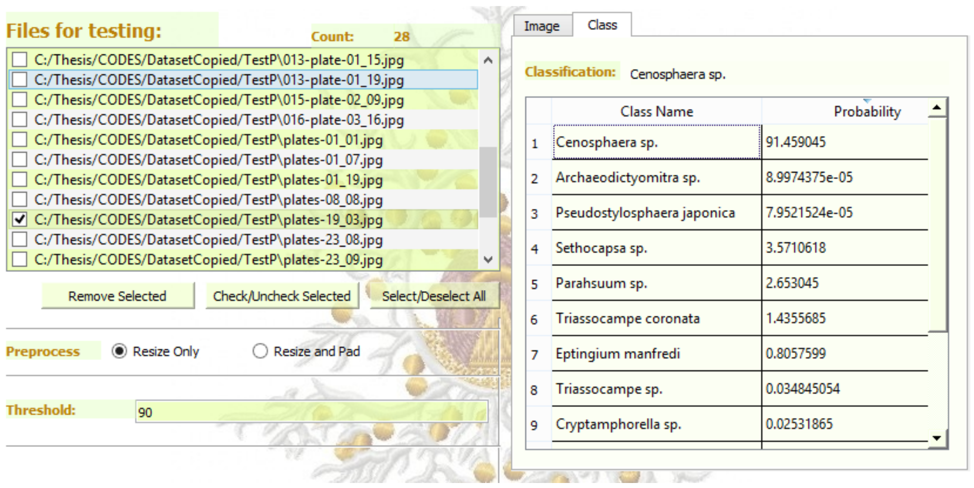
Just like the training images, the images that are needed to be classified will also undergo some preprocessing for the classification to be more accurate. The user gets to choose whether the images are going to be resized only or be pad around to maintain aspect ratio.

The best model that was trained on the training module will be automatically loaded to the classifier application and this is what the application will use to classify the images. The threshold value is the minimum probability that the classification should reach for an image to be classified to a class. If the highest probability did not reach the threshold value, then the class of the image will be set to unknown. This value can be set by the user as well.

Threshold: 90

Figure 37. RaDSS v02 Classifier Application Classification Specifications

The checked items are the only images that will be classified aftre clicking the Classify Selected Button. This is to ensure that the previously classified images will not be reclassified (with the same output) to reduce execution time and redundancy. After the classification is finished, the class tab beside the preview will be updated with the class name and the corresponding probability on the classes.



- [ ] C:/Thesis/CODES/DatasetCopied/TestP\013-plate-01_15.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\013-plate-01_19.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\015-plate-02_09.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\016-plate-03_16.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_01.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_07.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-01_19.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-08_08.jpg
- [x] C:/Thesis/CODES/DatasetCopied/TestP\plates-19_03.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-23_08.jpg
- [ ] C:/Thesis/CODES/DatasetCopied/TestP\plates-23_09.jpg

Preprocess: [✓ Resize Only] Resize and Pad

Threshold: 90

Classification: Cenosphaera sp.

	Class Name	Probability
1	Cenosphaera sp.	91.459045
2	Archaeodictyomitra sp.	8.9974375e-05
3	Pseudostylosphaera japonica	7.9521524e-05
4	Sethocapsa sp.	3.5710618
5	Parahsuum sp.	2.653045
6	Triassocampe coronata	1.4355685
7	Eptingium manfredi	0.8057599
8	Triassocampe sp.	0.034845054
9	Cryptamphorella sp.	0.02531865

Figure 38. RaDSS v02 Classifier Application Classification Results

The user can export clasification summary and details by clicking File menu and then Export result to PDF

More information about the different modules and how to use the application can be found in the User's Manual and can be viewed by clicking ”Help/Tutorials” under ”Help” menu as well.

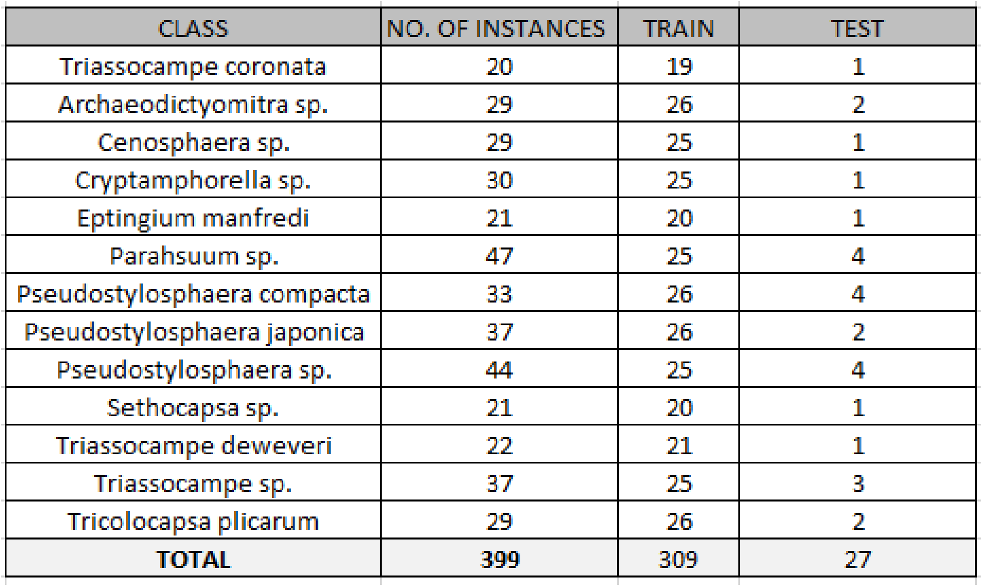
VI. Discussion

A. Dataset

The dataset comprises a total of 3,820 microfossil images of various Radiolarian species and were used to test the application. These images were provided by Professor Edanjarlo Marquez which are in the form of digital and printed (which were scanned).

Preprocessing The only preprocessing performed on the data is rescaling and can be done using the python's OpenCV library.

Classes Since the Radiolarian classes present on the dataset have unequal number of instances with the minimum of 4 and a maximum of 47, we have to reduce the classes so to have a minimum of 20 intances per class. The table below shows thirteen classes that is supported by the application and their correspoding class count used for training.

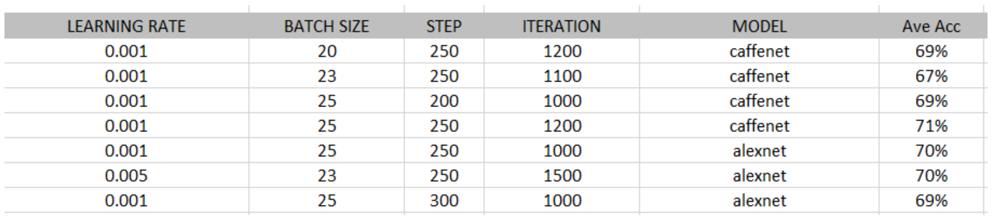


CLASS	NO. OF INSTANCES	TRAIN	TEST
Triassocampe coronata	20	19	1
Archaeodictyomitra sp.	29	26	2
Cenosphaera sp.	29	25	1
Cryptamphorella sp.	30	25	1
Eptingium manfredi	21	20	1
Parahsuum sp.	47	25	4
Pseudostylosphaera compacta	33	26	4
Pseudostylosphaera japonica	37	26	2
Pseudostylosphaera sp.	44	25	4
Sethocapsa sp.	21	20	1
Triassocampe deweveri	22	21	1
Triassocampe sp.	37	25	3
Tricolocapsa plicarum	29	26	2
TOTAL	**399**	309	27

Figure 39. RaDSS v02 Dataset List

The validation set will be used for cross-validation during training to test if the model performs well and obtains a high accuracy. Caffe automatically do this by just specifying the validation set.

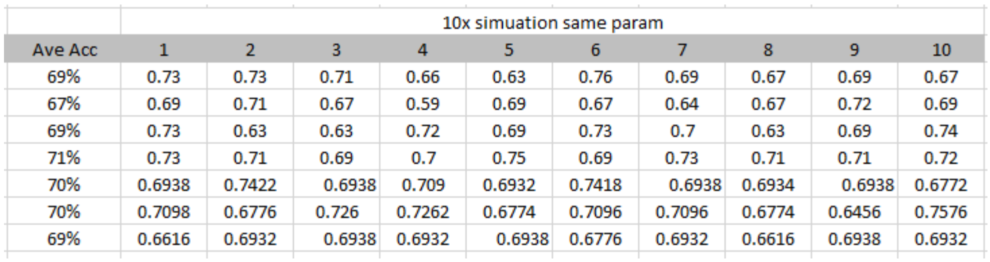
In order to obtain the best accuracy, we have to tweak the hyperparameters of the network: adjusting the learning rate, batch size, step and number of iteration.



LEARNING RATE	BATCH SIZE	STEP	ITERATION	MODEL	Ave Acc
0.001	20	250	1200	caffenet	69%
0.001	23	250	1100	caffenet	67%
0.001	25	200	1000	caffenet	69%
0.001	25	250	1200	caffenet	71%
0.001	25	250	1000	alexnet	70%
0.005	23	250	1500	alexnet	70%
0.001	25	300	1000	alexnet	69%

Figure 40. RaDSS v02 Various accuracy with different values of hyperparameter

Since the accuracy varies because of random sampling, these accuracies were obtained through averaging 10 training runs on the same values of hyperparameters.



	10x simuation same param									
Ave Acc	1	2	3	4	5	6	7	8	9	10
69%	0.73	0.73	0.71	0.66	0.63	0.76	0.69	0.67	0.69	0.67
67%	0.69	0.71	0.67	0.59	0.69	0.67	0.64	0.67	0.72	0.69
69%	0.73	0.63	0.63	0.72	0.69	0.73	0.7	0.63	0.69	0.74
71%	0.73	0.71	0.69	0.7	0.75	0.69	0.73	0.71	0.71	0.72
70%	0.6938	0.7422	0.6938	0.709	0.6932	0.7418	0.6938	0.6934	0.6938	0.6772
70%	0.7098	0.6776	0.726	0.7262	0.6774	0.7096	0.7096	0.6774	0.6456	0.7576
69%	0.6616	0.6932	0.6938	0.6932	0.6938	0.6776	0.6932	0.6616	0.6938	0.6932

Figure 41. RaDSS v02 Average accuracies with same values of hyperparameter

VII. Conclusion

RaDSS v02 is an application that will help the micropaleontologist in classifying Radiolarian species. It is made possible by Deep Learning algorithms, in this case, the convolutional neural network. With miniman preprocessing, it is able to analize raw Radiolarian microfossil image and tweaking the hyperparameters might improve the accuracy. The result containing the information about the classification can be exported through a PDF.

VIII. Recommendation

RaDSS v02's accuracy will greatly increase if there are more data and instances per class. The more data in a convolutional neural network, the more learning it will get.

Also, some heavy architectures like googleNet or ResNet might perform better than the ones availbale in RaDSS v02. Augmentation will also help increase the number of data and will ensure that the model performs well even with the different orientation of an image.

IX. Bibliography

[1] "Radiolaria — new world encyclopedia," *New World Encyclopedia*, 2015. n.p. Web. 04 January 2018.

[2] M. Asaravala, H. Lam, S. Litty, J. Phillips, and T.-T. Wu, "Introduction to the radiolaria," 2000. n.p. Web. 04 January 2018.

[3] H. M. Bik, "Lets rise up to unite taxonomy and technology," *PLOS Biology*, vol. 15, pp. 1–4, 08 2017.

[4] A. van den Oord, I. Korshunova, J. Burms, J. Degrave, L. Pigou, P. Buteneers, and S. Dieleman, "Classifying plankton with deep neural networks," March 2015. n.p. Web. 04 January 2018.

[5] J. Dai, R. Wang, H. Zheng, G. Ji, and X. Qiao, "Zooplanktonet: Deep convolutional network for zooplankton classification," in *OCEANS 2016 - Shanghai*, pp. 1–6, April 2016.

[6] L. A. Apostol, E. Marquez, P. Gasmen, and G. Solano, "Radss: A radiolarian classifier using support vector machines," *2016 7th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pp. 1–6, 2016.

[7] C. Fimbres-Castro, J. Ãlvarez-Borrego, I. Vazquez-Martinez, T. L. Espinoza-Carreon, A. E. Ulloa-Perez, and M. A. Bueno-Ibarra, "Nonlinear correlation by using invariant identity vectors signatures to identify plankton," *Gayana (Concepcion)*, vol. 77, pp. 105 – 124, 00 2013.

[8] A. S. Keçeli, A. Kaya, and S. U. Keçeli, "Classification of radiolarian images with hand-crafted and deep features," *Computers and Geosciences*, vol. 109, pp. 67–74, Dec. 2017.

[9] J. H. Lipps, "Microfossils," n.p. n.d. Web. 04 January 2018.

[10] P. A. Swaby, "Vides: an expert system for visually identifying microfossils," *IEEE Expert*, vol. 7, pp. 36–42, April 1992.

[11] M. A. O'Neill and M. Denos, "Automating biostratigraphy in oil and gas exploration: Introducing geodaisy," *Journal of Petroleum Science and Engineering*, vol. 149, no. Supplement C, pp. 851 – 859, 2017.

[12] D. J. Cindy M. Wong, "Dynamic hierarchical algorithm for accelerated microfossil identification," *Proceedings SPIE Image Processing: Machine Vision Applications VIII*, vol. 9405, pp. 1 – 15, 2015.

[13] L. Tindall, C. Luong, and A. Saad, "Plankton classification using vgg16 network," 2015.

[14] P. Jindal and R. Mundra, "Plankton classification using hybrid convolutional network-random forests architectures,"

[15] Y. Kuang, "Deep neural network for deep sea plankton classification," 2015.

[16] S. Liu, M. Thonnat, and M. Berthod, "Automatic classification of planktonic foraminifera by a knowledge-based system," in *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, pp. 358–364, Mar 1994.

[17] A. Pedraza, G. Bueno, O. Deniz, G. Cristbal, S. Blanco, and M. Borrego-Ramos, "Automated diatom classification (part b): A deep learning approach," *Applied Sciences*, vol. 7, no. 5, 2017.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[19] M. D. Zeiler and R. Fergus, *Visualizing and Understanding Convolutional Networks*, pp. 818–833. Cham: Springer International Publishing, 2014.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[23] "Radiolaria," *Miracle*, 2002. n.p. Web. 04 January 2018.

[24] "What are radiolarians," n.p. n.d. Web. 04 January 2018.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[26] A. Deshpande, "A beginner's guide to understanding convolutional neural network," n.p. n.d. Web. 04 January 2018.

[27] U. Karn, "An intuitive explanation of convolutional neural networks," August 2016. n.p. Web. 04 January 2018.

[28] "Caffe," n.p. n.d. Web. 04 January 2018.

X. Appendix

A. Forms

B. Source Code

```
import settings
import sys
import os
import cv2
import numpy as np
import glob
import random
import caffe
import time
import subprocess
import datetime
import plot_learning_curve as plc
from PyQt5.uic import loadUi
from PyQt5 import QtGui, QtCore
from readFiles import FileReader
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from collections import Counter
from augmentor import augment
from PIL import Image
from create_lmdb import CreateLMDB
from functools import partial
from preprocess import PreprocessImage as pr
from create_html import createHTML

class MainApp(QMainWindow):
        def __init__(self):
                super(MainApp, self).__init__()
                self.show_home()

        def show_home(self):
                loadUi(settings.ui_path + 'mainApp.ui', self)
                self.setWindowTitle("RaDSS V02")
                self.setWindowIcon(QtGui.QIcon(settings.icon_path))

                # -- code below connects action to the buttons/menu/etc
                self.trainBtn.clicked.connect(self.show_training_app)
                self.classBtn.clicked.connect(self.show_test_app)

                self.actionLoad_Manual.triggered.connect(self.loadManual)

        def clearLayout(layout):
                while layout.count():
                        child = layout.takeAt(0)
                        if child.widget() is not None:
                                child.widget().deleteLater()
                        elif child.layout() is not None:
                                clearLayout(child.layout())
        def show_training_app(self):

                loadUi(settings.ui_path + 'trainApp.ui', self)
                self.setWindowTitle("RaDSS V02 Training Module")
                self.setWindowIcon(QtGui.QIcon(settings.icon_path))

                # INSTANCE VARIABLES
                self.label_map = []
                self.classes = []
                self.classCount = []
                self.architecture = "caffenet"
                self.currentItem = None

                # -- code below connects action to the buttons/menu/etc
                self.trainButton.clicked.connect(self.train_data_method)
                #self.trainButton.clicked.connect(self.train_data_temp)
                self.removeButton.clicked.connect(self.remove_selected)
                self.selUnlabeled.clicked.connect(self.select_all_unlabeled)

                #don't remove menu actions! for example
                self.actionAdd_Images.triggered.connect(self.add_images)
                self.actionAdd_Image.triggered.connect(self.add_image)
                self.actionEdit_Configuration.triggered.connect(self.edit_configuration)
                self.actionExport_labels.triggered.connect(self.export_labels)
                self.actionBack_to_Home.triggered.connect(self.show_home)
                self.actionShow_classes.triggered.connect(self.showClasses)
                self.actionLoad_Manual.triggered.connect(self.loadManual)

                self.list_training_images.itemClicked.connect(self.image_clicked_preview)
                self.image_classification.returnPressed.connect(self.classification_changed)

                # validators
                self.batch_size.setValidator(QIntValidator())
                self.iter_no.setValidator(QIntValidator())
                self.train_step.setValidator(QIntValidator())

                # set initial fields
                self.filecnt.setText("0")
                self.clscnt.setText("0")

        def show_test_app(self):
                loadUi(settings.ui_path + 'testApp.ui', self)
                self.setWindowTitle("RaDSS V02 Classifier Application")
                self.setWindowIcon(QtGui.QIcon(settings.icon_path))

                # Instance Variables
                self.classes = np.load(settings.data_path + "classDict.npy")
                self.currentItem = None
                self.predictions = []

                # one dimensional array that corresponds to the indices of list_testing_images
                self.probab = []
                self.thresh = 90.

                # -- code below connects action to the buttons/menu/etc

                self.classifyButton.clicked.connect(self.test_data_method)
                self.selectAllBtn.clicked.connect(self.select_all)
                self.chkSelectedBtn.clicked.connect(self.check_selected)
                self.btnRemove.clicked.connect(self.test_remove_selected)

                #don't remove menu actions! for example
                self.actionAdd_Images_Test.triggered.connect(self.test_add_images)
                self.actionAdd_Image_Test.triggered.connect(self.test_add_image)
                self.actionExport_to_PDF.triggered.connect(self.exportToPDF)
                self.actionBack_to_Home.triggered.connect(self.show_home)
                self.actionLoad_Manual.triggered.connect(self.loadManual)

                self.list_testing_images.itemClicked.connect(self.test_image_clicked_preview)
                self.threshold.returnPressed.connect(self.threshold_changed)

                # populate table with the classes
                for row, cls in enumerate(self.classes):
                        self.tableWidget.insertRow(row)
                        self.tableWidget.setItem(row , 0, QTableWidgetItem(cls))

                print("Model to use: ", settings.classifier_model)

        ############################################################
        ############################################################
        ############################################################
        ## TRAIN MODULE METHODS ####################################

        # creates an absolute_filename-label array self.train and self.val
        @pyqtSlot()
        def train_data_method(self):
                print("Train")

                if self.validate_inputs() == False:
                        return

                self.completed = 0

                # default is caffenet; check if alexNet is chosenn
                if self.alexNet.isChecked():
                        self.architecture = "alexnet"

                # save class dictionary
                self.count_classes()
                np.save(settings.data_path + "classDict.npy", self.classes)

                # edit prototxt files based on hyperparameters set by user
                self.update_progress_bar("Creating solver pototxts...", 1)
                self.edit_prototxts()

                # shuffle the list (this is 2 dimensional so use np)
                # checked and working
                #print(*self.label_map, sep='\n')
                np.random.shuffle(self.label_map)

                # working
                # divide train and val randomly
                self.update_progress_bar("Dividing train and val set...", 2)
                self.train = []
                self.val = []
                for i in range(len(self.label_map)):
                        temp = self.label_map[i][:]                    # [:] is necessary to pass by value!!
```

```
                        temp[1] = self.get_image_num_label(temp[1])
                        # put all that is divisible by five to the val set, this will comprise 20% of the data
                        if(i % 5 == 0):
                                self.val.append(temp)
                        else:
                                self.train.append(temp)
                # mappings
                # array: image[0] with label: label[0]
                self.train_data = []
                self.train_label = []
                self.val_data = []
                self.val_label = []

                self.update_progress_bar("Converting Images to Array...", 2)

                # Working!: Convert and Augment
                # convert each TRAIN filenames to nparray and augment
                for imahe in self.train:
                        # -- optional methods but will resort to just resize for now
                        # -- these can be applied to val as well
                        #img = pr.transform_img(imahe)
                        #img = pr.resizeAndPad(img)
                        #cv2.imshow("imahe", img)

                        # automatic type is uint8 <-- imgaug requirement
                        img = cv2.imread(imahe[0], cv2.IMREAD_COLOR)

                        if self.resizeRadio.isChecked():
                                img = cv2.resize(img, (settings.IMAGE_HEIGHT, settings.IMAGE_WIDTH))
                                #cv2.imshow("imahe", img)
                        else:
                                img = pr.resizeAndPad(img)
                                #cv2.imshow("imahe", img)

                        self.train_data.append(img)
                        # augment image and append augmented images on train_data
                        #self.train_data.extend(augment(img, 10))
                        # append the original's image label and 10 augmented of the same label
                        #self.train_label.extend([imahe[1]] * 11)
                        self.train_label.append(imahe[1])               # remove when augment is on

                # convert each VAL filenames to nparray and augment
                for imahe in self.val:
                        img = cv2.imread(imahe[0], cv2.IMREAD_COLOR)

                        if self.resizeRadio.isChecked():
                                img = cv2.resize(img, (settings.IMAGE_HEIGHT, settings.IMAGE_WIDTH))
                        else:
                                img = pr.resizeAndPad(img)

                        self.val_data.append(img)
                        # augment image and append augmented images on val_data
                        #self.val_data.extend(augment(img, 10))
                        # append the original's image label and 10 augmented of the same label
                        #self.val_label.extend([imahe[1]] * 11)
                        self.val_label.append(imahe[1])         # remove when augment is on

                # create_lmdb
                self.update_progress_bar("Create LMDB Files...", 5)
                lmdb = CreateLMDB(self.train_data, self.train_label, self.val_data, self.val_label)

                # image_mean
                # remove last slashes on paths
                # make_imagenet_mean.sh lmdb_path image_mean_path caffe_tools_path
                # syntax: ./compute_image_mean path/to/train_lmdb path/to/images_mean.binaryproto
                self.update_progress_bar("Create Image Mean File...", 1)
                os.system(settings.tools_path + "compute_image_mean " + settings.database_path +
                "train_lmdb " +  settings.data_path + "images_mean.binaryproto")

                self.update_progress_bar("Training Network...", 10)

                # train
                # syntax: ./caffe train -solver path/to/solver.prototxt 2>&1 | tee /path/to/logs/
                now = datetime.datetime.now().strftime("%Y%m%d_%H%M")
                self.logfile = settings.data_path + "logs/"+ self.architecture + "_" + str(now) + "_log.log"
                self.plotFile = settings.data_path + "plots/"+ self.architecture + "_" + str(now) + "_plot"
                os.system(settings.tools_path + "caffe train -solver " + settings.models_path +
                self.architecture + "/solver.prototxt 2>&1 | tee " + self.logfile)

                self.update_progress_bar("Training Finished! Caffe Models saved at " +
                settings.models_path + self.caffemodel_name, 79)

                self.display_training_results()

        # edit the chosen architecture's prototxt based on user hyperparameter and some settings
        def edit_prototxts(self):

                # hyperparameters
                bs = (int)(self.batch_size.text())
                lr = (float)(self.learning_rate.text())
                stepsize = (int)(self.train_step.text())
                max_iter = (int)(self.iter_no.text())
                #solver_mode: CPU = 0, GPU != 0
                mode = 0

                reader = FileReader("dummyfile")
                print("Edit Train Prototxt of ", self.architecture)
                reader.edit_train_prototxt(self.architecture, bs, len(self.classes))
                print("Create Solver Prototxt of ", self.architecture)
                # retrieve the filename of the saved model
                self.caffemodel_name = reader.solver(self.architecture, lr, stepsize, max_iter, mode)
                self.caffemodel_name = self.caffemodel_name + "iter_" + str(max_iter) + ".caffemodel"
                print("Edit Deploy Prototxt of ", self.architecture)
                reader.deploy(self.architecture, len(self.classes))

        def validate_inputs(self):
                # don't continue if there is no data
                if len(self.label_map) == 0:
                        self.statusBar().showMessage("No Data Found. Please Add Images!")
                        return False

                try:
                        if((float)(self.learning_rate.text()) >= 1):
                                self.statusBar().showMessage("Please input valid learning rate.")
                                return False
                except: # string
                        self.statusBar().showMessage("Please input valid learning rate.")
                        return False

                # check if all are labeled
                lst = -1
                for i in range(len(self.label_map)):
                        if(self.label_map[i][1] == None):
                                self.statusBar().showMessage("Please put labels on all images!")
                                return False

                return True

        def display_training_results(self):
                print("Display Results")
                # plot curves argument: log_path, output_path, plot_path
                self.accuracy = plc.plot_curves(self.logfile, settings.data_path + "logs/", self.plotFile)

                self.trainResultsDiag = QDialog()
                self.trainResultsDiag.setWindowIcon(QtGui.QIcon(settings.icon_path))
                loadUi(settings.ui_path + 'plotGui.ui', self.trainResultsDiag)
                self.trainResultsDiag.setWindowModality(QtCore.Qt.ApplicationModal)
                self.trainResultsDiag.show()
                self.trainResultsDiag.exportClassifierModel.clicked.connect(
                self.export_caffemodel_to_classifier_app)

                # display plot
                pixmap = QtGui.QPixmap(self.plotFile)
                pixmap = pixmap.scaled(self.trainResultsDiag.plotPreview.width(),
                self.trainResultsDiag.plotPreview.height(), QtCore.Qt.KeepAspectRatio)
                self.trainResultsDiag.plotPreview.setPixmap(pixmap)

                # display accuracy
                self.trainResultsDiag.accuracy.setText(str(self.accuracy))

                # display caffemodel trained and saved
                self.trainResultsDiag.caffemodel.setText(str(self.caffemodel_name))

        def export_caffemodel_to_classifier_app(self):
                print("Export model to Classifier Application.")
                settings.classifier_model = self.caffemodel_name
                self.trainResultsDiag.close()
                self.statusBar().showMessage(settings.models_path + self.caffemodel_name +
                " was exported to the Classifier Application")

        ########################################################################
        # GUI Methods like classification changed, and image clicked

        def classification_changed(self):
                print("Class changed")
                idx = self.list_training_images.currentRow()
                self.statusBar().showMessage("Classification Changed for " +
                self.list_training_images.item(idx).text())
                self.label_map[idx][1] = self.image_classification.text()
                self.image_classification.clearFocus()

        def image_clicked_preview(self, item):
                self.image_classification.setEnabled(True)

                if(self.currentItem == item):
                        item.setSelected(False)
                        self.currentItem = None
                        self.imagePreview.setText("No Image Selected")
                        self.image_classification.setEnabled(False)
                        return

                # preview
                pixmap = QtGui.QPixmap(str(item.text()))
                pixmap = pixmap.scaled(self.imagePreview.width(),
                self.imagePreview.height(), QtCore.Qt.KeepAspectRatio)
                self.imagePreview.setPixmap(pixmap)

                # set label on click
                idx = self.list_training_images.currentRow()
                item = self.label_map[idx][1]
                if(item == None):
                        self.image_classification.setPlaceholderText("None")
                        self.image_classification.setText("")
                else:
                        self.image_classification.setText(item)

                self.currentItem = item

        def update_progress_bar(self, msg, num):
                self.statusBar().showMessage(msg)
                self.completed += num
                self.progressBar.setValue(self.completed)

        def remove_selected(self):
                for it in self.list_training_images.selectedItems():
                        del self.label_map[self.list_training_images.row(it)]
                        # remove class pa if that item is the only one with that class!
                        self.list_training_images.takeItem(self.list_training_images.row(it))
                self.update_gui_fields()

        # working
        def select_all_unlabeled(self):
                self.list_training_images.clearSelection()
                lst = -1
                for i in range(len(self.label_map)):
                        if(self.label_map[i][1] == None):
                            self.list_training_images.item(i).setSelected(True)
                            lst = i
                if(lst == -1):
                        self.statusBar().showMessage("No more unlabeled images!")
                else:
                        self.list_training_images.setCurrentRow(lst)

        # used by add_images, add_image and remove_selected
        def update_gui_fields(self):
                self.filecnt.setText(str(self.list_training_images.count()))
                self.clscnt.setText(str(self.count_classes()))

        def export_labels(self):
                if self.list_training_images.count() == 0:
                        self.statusBar().showMessage("No labels to export!")
                        return
                reader = FileReader("dummyfile")
                fileName = reader.export_to_excel(self, self.label_map)
                if fileName:
                        self.statusBar().showMessage("Saved file to " + fileName)

        def export_classes(self):
                reader = FileReader("dummyfile")
                fileName = reader.export_to_excel(self, self.classCount)
                if fileName:
                        self.statusBar().showMessage("Saved file to " + fileName)

        def showClasses(self):
                print("Show Classes")
                self.showClassesDiag = QDialog()
                self.showClassesDiag.setWindowIcon(QtGui.QIcon(settings.icon_path))
                loadUi(settings.ui_path + 'classDict.ui', self.showClassesDiag)
                self.showClassesDiag.setWindowModality(QtCore.Qt.ApplicationModal)
                self.showClassesDiag.show()

                row = 0
                for cls in self.classCount:
                        self.showClassesDiag.classTable.insertRow(row)
                        self.showClassesDiag.classTable.setItem(row , 0, QTableWidgetItem(cls[0]))
                        self.showClassesDiag.classTable.setItem(row , 1, QTableWidgetItem(str(cls[1])))
                        row+=1


        ######### settings method
        def edit_configuration(self):
                self.setting = QDialog()
                self.setting.setWindowIcon(QtGui.QIcon(settings.icon_path))
                loadUi(settings.ui_path + 'trainSettings.ui', self.setting)
                self.setting.setWindowModality(QtCore.Qt.ApplicationModal)
                self.setting.show()

                # put based on settings file
                self.setting.databaseFolder.setText(settings.database_path)
                self.setting.dataFolder.setText(settings.data_path)
                self.setting.modelsFolder.setText(settings.models_path)

                self.setting.browsedbFolder.clicked.connect(partial(self.select_directory,
                self.setting.databaseFolder))
                self.setting.browsedataFolder.clicked.connect(partial(self.select_directory,
                self.setting.dataFolder))
                self.setting.browsemodelsFolder.clicked.connect(partial(self.select_directory,
                self.setting.modelsFolder))

                self.setting.btnOk.clicked.connect(self.settings_okay_clicked)

        @pyqtSlot()
        def settings_okay_clicked(self):
                settings.database_path = self.settings.databaseFolder.text()
                settings.data_path = self.settings.dataFolder.text()
                settings.models_path = self.settings.modelsFolder.text()
                self.setting.close()

        ############ --end settings method

        def select_directory(self, lineEdit):
                file = str(QFileDialog.getExistingDirectory(self, "Select Directory"))
                if file:
                        lineEdit.setText(file)
                        # checkings

        @pyqtSlot()
        def select_file(self):
                options = QFileDialog.Options()
                options |= QFileDialog.DontUseNativeDialog
                file, _ = QFileDialog.getOpenFileName(self,"Open File", "",
                "Excel Files (*.xls *.xlsx) ;; CSV Files (*.csv)", options=options)
                if file:
                        print("Add Images File")
                        self.addImagesDialog.trainFile.setText(file)
                        # checkings



        ####################################################################
        # --- DATA PROCESSING methods
        # count classes, count models to use
        # preprocessing to use

        # working!
        def count_classes(self):
                if self.list_training_images.count() == 0:
                        return 0

                # reset!
                self.classes = []
                self.classCount = []
                arr = np.array(self.label_map)
                count = Counter(arr[:,1])
                # Accessing each element:
                for k,v in count.items():
                        if(k != None):
                                self.classes.append(k)
                                self.classCount.append([k, v])
                        #or count['itemname']
                self.classes = sorted(self.classes)
                self.classCount = sorted(self.classCount)

                return len(self.classes)

        ####################################################################
        # --- general methods

        # get image label from the text file given by the user
        # used by add_images_to_training
        def get_image_label(self, imgpath, file):

                for i in range(len(file)):
                        if(file[0][i] in imgpath):
                                return file[1][i]
                return None

        # returns the number representation of a label in string
        def get_image_num_label(self, strlabel):
                return self.classes.index(strlabel)

        def check_file_duplicates(self):
                pass

        ####################################################################
        # --- add images methods

        # QDialog things
        # the real method is when OK is clicked --> add_images_to_training()
        def add_images(self):
                self.addImagesDialog = QDialog()
                self.addImagesDialog.setWindowIcon(QtGui.QIcon(settings.icon_path))
                loadUi(settings.ui_path + 'trainAddImages.ui', self.addImagesDialog)
                self.addImagesDialog.setWindowModality(QtCore.Qt.ApplicationModal)
                self.addImagesDialog.show()

                self.addImagesDialog.trainFolder.setText
                # partial(method, argv)
```

```
            self.addImagesDialog.browseTrainDir.clicked.connect(partial(self.select_directory,
            self.addImagesDialog.trainFolder))

            self.addImagesDialog.browseFile.clicked.connect(self.select_file)
            self.addImagesDialog.btnOk.clicked.connect(self.add_images_to_training)
            self.addImagesDialog.btnCancel.clicked.connect(self.cancel_clicked)

    @pyqtSlot()
    def cancel_clicked(self):
            # just close and do nothing
            self.addImagesDialog.close()

    # ----------------------------------
    # self.label_map changed
    @pyqtSlot()
    def add_images_to_training(self):
            # add checkings if all are directories before closing
            self.addImagesDialog.close()
            print("Add Images Closed!")

            reader = FileReader(self.addImagesDialog.trainFile.text())

            #based on labels file but this is tuple with labels
            addfileLabels = reader.get_data_and_labels()
            addfileNames = glob.glob(self.addImagesDialog.trainFolder.text() + "/**/*.jpg", recursive=True)

            self.completed = 0
            factor = 100/len(addfileNames)
            self.statusBar().showMessage("Processing images...")
            # append on gui those that are in dir and subdir
            for file in addfileNames:
                    self.update_progress_bar("Processing image " + file, factor)
                    item = QListWidgetItem(file)
                    self.list_training_images.addItem(item)
                    label = self.get_image_label(file, addfileLabels)

                    # append on my DATA
                    self.label_map.append([file, label])

            self.progressBar.setValue(100)
            self.statusBar().showMessage("Processing images finished!!!")
            self.update_gui_fields()
            self.progressBar.setValue(0)


    ###################################################################
    # --- add image method

    # ----------------------------------
    # self.label_map append with None
    def add_image(self):
            options = QFileDialog.Options()
            options |= QFileDialog.DontUseNativeDialog
            file, _ = QFileDialog.getOpenFileName(self, "Open File", "",
            "Image Files (*.jpg)", options=options)
            if file:
                    print("Add Images")
                    # checkings pa

                    # append on gui
                    item = QListWidgetItem(file)
                    self.list_training_images.addItem(item)
                    self.label_map.append([file, None])

                    self.update_gui_fields()

    ####################################################################
    ####################################################################
    ####################################################################
    ##### CLASSIFIER APP ###############################################

    ####################################################################
    # GUI METHOD
    def test_image_clicked_preview(self, item):
            # reset table values with class names: alphabetical and no values yet
            self.tableWidget.setSortingEnabled(0)
            for i in range(len(self.classes)):
                    self.tableWidget.setItem(i , 0, QTableWidgetItem(""))
                    self.tableWidget.setItem(i , 1, QTableWidgetItem(""))

            for i in range(len(self.classes)):
                    self.tableWidget.setItem(i , 0, QTableWidgetItem(self.classes[i]))

            if(self.currentItem == item):
                    item.setSelected(False)
                    self.currentItem = None
                    self.imagePreview.setText("No Image Selected")
                    self.classification.setText("None")
                    return

            # preview
            pixmap = QtGui.QPixmap(str(item.text()))
```

```
        pixmap = pixmap.scaled(self.imagePreview.width(), self.imagePreview.height(),
        QtCore.Qt.KeepAspectRatio)

        self.imagePreview.setPixmap(pixmap)

        # set current item for select/deselect
        self.currentItem = item

        # Set classification
        cls = self.probab[self.list_testing_images.currentRow()]
        if((cls == 0).sum() != len(self.classes)):
                if cls[cls.argmax()] >= self.thresh: # only classify if >= threshold
                        self.classification.setText(str(self.classes[cls.argmax()]))
                else:
                        self.classification.setText("UNKNOWN")
                for i, prob in enumerate(cls):
                        self.tableWidget.setItem(i , 1, QTableWidgetItem(str(prob)))
        else:
                self.classification.setText("None")

        self.tableWidget.setSortingEnabled(1)

def threshold_changed(self):
        self.thresh = (float)(self.threshold.text())
        self.statusBar().showMessage("Threshold Changed!")
        self.threshold.clearFocus()

#select/deselect all
@pyqtSlot()
def select_all(self):
        #if(list_testing_images.selectedItems() > 0)
        check = QtCore.Qt.Checked
        if(self.list_testing_images.item(0).checkState() == QtCore.Qt.Checked):
                check = QtCore.Qt.Unchecked
        for i in range(self.list_testing_images.count()):
                self.list_testing_images.item(i).setCheckState(check)

@pyqtSlot()
def check_selected(self):
        for item in self.list_testing_images.selectedItems():
                if(item.checkState() == QtCore.Qt.Unchecked):
                        item.setCheckState(QtCore.Qt.Checked)
                else:
                        item.setCheckState(QtCore.Qt.Unchecked)

def test_remove_selected(self):
        for it in self.list_testing_images.selectedItems():
                del self.probab[self.list_testing_images.row(it)]
                self.list_testing_images.takeItem(self.list_testing_images.row(it))

def exportToPDF(self):
        if(len(self.wdclass) == 0):
                self.statusBar().showMessage("No Classified Images yet! Export Failed.")
                return

        html = createHTML(settings.data_path + "myPage.html", self.classes)
        html.createHeader(settings.classifier_model, self.threshold.text())
        for idx, prb in enumerate(self.probab):
                if((prb == 0).sum() != len(self.classes)):
                        html.createImg(self.list_testing_images.item(idx).text(), prb,
                        self.classes[prb.argmax()], "luh")
        html.createEnd()
        fileName = html.save_pdf(self)

        self.statusBar().showMessage("Export Finished! Result saved in " + fileName)

########################################################################
# TEST METHOD

@pyqtSlot()
def test_data_method(self):
        print("Test")

        # reset the list to test and the list of indices to test, respectively
        self.test = []
        self.wdclass = []

        self.statusBar().showMessage("Testing...")
        # no need to create lmdb because test directory is enough

        for i in range(self.list_testing_images.count()):
                item = self.list_testing_images.item(i)
                if(item.checkState() == QtCore.Qt.Checked):
                        img = cv2.imread(item.text(), cv2.IMREAD_COLOR)

                        if self.resizeRadio.isChecked():                #resizeOnly
                                img = cv2.resize(img, (settings.IMAGE_HEIGHT, settings.IMAGE_WIDTH))
                                #cv2.imshow("imahe", img)
                        else:                                           #resize and pad
                                img = pr.resizeAndPad(img)
                                #cv2.imshow("imahe", img)
                        self.test.append(img)
                        self.wdclass.append(i)          # appends the index to the list with classes

        if(len(self.test) == 0):
                print("No Test Images Found!")
                self.statusBar().showMessage("No Test Images Selected!")
                return

        # convert images to .npy for input in classify.py
        np.save(settings.data_path + "test_input.npy", self.test)

        # convert train binaryproto to npy for caffe mean-file of classify,py
        blob = caffe.proto.caffe_pb2.BlobProto()
        data = open(settings.data_path + "images_mean.binaryproto" , 'rb' ).read()
        blob.ParseFromString(data)
        arr = np.array( caffe.io.blobproto_to_array(blob) )
        out = arr[0]
        np.save( settings.data_path + "images_mean.npy" , out )

        # classify
        mod = settings.classifier_model.split("_")[0]

        self.classify_method(settings.data_path + "test_input.npy", settings.data_path +
        "test_output.npy", settings.models_path + mod + "/deploy.prototxt", settings.models_path +
        settings.classifier_model, settings.data_path + "images_mean.npy", False)

        self.storeProbabsAfterTest()

        self.statusBar().showMessage("Classification Finished!")

def classify_method(self, input_file, output_file, model_def, pretrained_model, mean_file,
gpu=True, center_only=False, images_dim = '256,256', input_scale=None, raw_scale=255.0,
channel_swap='2,1,0', ext='jpg'):

        image_dims = [int(s) for s in images_dim.split(',')]

        mean, channel_swap = None, None
        if mean_file:
                mean = np.load(mean_file)
        if channel_swap:
                channel_swap = [int(s) for s in args.channel_swap.split(',')]

        if gpu:
                caffe.set_mode_gpu()
                print("GPU mode")
        else:
                caffe.set_mode_cpu()
                print("CPU mode")

        classifier = caffe.Classifier(model_def, pretrained_model,
                image_dims=image_dims, mean=mean,
                input_scale=input_scale, raw_scale=raw_scale, channel_swap=channel_swap)

        # Load numpy array (.npy), directory glob (*.jpg), or image file.
        input_file = os.path.expanduser(input_file)
        if input_file.endswith('npy'):
                print("Loading file: %s" % input_file)
                inputs = np.load(input_file)
        elif os.path.isdir(input_file):
                print("Loading folder: %s" % input_file)
                inputs =[caffe.io.load_image(im_f)
                                for im_f in glob.glob(input_file + '/*.' + ext)]
        else:
                print("Loading file: %s" % input_file)
                inputs = [caffe.io.load_image(input_file)]

        print("Classifying %d inputs." % len(inputs))

        # Classify.
        start = time.time()
        self.predictions = classifier.predict(inputs, not center_only)

        print("Done in %.2f s." % (time.time() - start))

        # Save
        print("Saving results into %s" % output_file)
        np.save(output_file, self.predictions)

def storeProbabsAfterTest(self):
        for indx, item in enumerate(self.wdclass):
                self.probab[item] = self.predictions[indx]
                self.probab[item][:] = [(x*100.) for x in self.probab[item]]


##########################################################################
# --- add images method

# ------------------------------
# self.probab changed
def test_add_images(self):
        file = str(QFileDialog.getExistingDirectory(self, "Select Directory"))
        if file:
                self.statusBar().showMessage("Processing images...")
                # append on gui those that are in files and subdir
                for file in glob.iglob(file + "/**/*jpg", recursive=True):
                        item = QListWidgetItem(file)
                        item.setFlags(item.flags() | QtCore.Qt.ItemIsUserCheckable)
                        item.setCheckState(QtCore.Qt.Unchecked)
                        self.list_testing_images.addItem(item)
                        self.probab.append(np.array([0]*len(self.classes)))

            self.filecnt.setText(str(self.list_testing_images.count()))
            self.statusBar().showMessage("Processing images finished!")

    ##############################################
    # --- add image method

    # --------------------------------
    # self.probab changed
    def test_add_image(self):
            options = QFileDialog.Options()
            options |= QFileDialog.DontUseNativeDialog
            file, _ = QFileDialog.getOpenFileName(self, "Open File", "",
            "Image Files (*.jpg *.png)", options=options)
            if file:
                    print("Add Image")

                    # append on gui
                    item = QListWidgetItem(file)
                    item.setFlags(item.flags() | QtCore.Qt.ItemIsUserCheckable)
                    item.setCheckState(QtCore.Qt.Unchecked)
                    self.list_testing_images.addItem(item)
                    self.probab.append(np.array([0]*len(self.classes)))

            self.filecnt.setText(str(self.list_testing_images.count()))

    ######################################################################
    ## -- USED BY BOTH TRAIN AND TEST APP -- ############################
    #####################################
    def loadManual(self):
            print("Load Help")
            self.statusBar().showMessage("Loading Help...")
            self.helpManual = QDialog()
            loadUi(settings.ui_path + 'helpManual.ui', self.helpManual)
            self.helpManual.setWindowTitle("RaDSS V02 Help Tutorial")
            self.helpManual.setWindowIcon(QtGui.QIcon(settings.icon_path))
            self.helpManual.setWindowModality(QtCore.Qt.ApplicationModal)
            self.helpManual.show()

            self.helpManual.treeWidget.itemClicked.connect(self.loadManualItem)

            self.statusBar().showMessage("")

    def loadManualItem(self, item):
            file = open(settings.help_path + item.text(0) + ".txt", "r", encoding="utf8")
            file_contents = file.read()
            self.helpManual.preview.setText(str(file_contents))
            file.close()

#main
app = QApplication(sys.argv)
main = MainApp()
main.show()
sys.exit(app.exec_())



import os
import glob
import random
import numpy as np

import cv2

import caffe
from caffe.proto import caffe_pb2
import lmdb
import settings

#Size of images
IMAGE_WIDTH = settings.IMAGE_WIDTH
IMAGE_HEIGHT = settings.IMAGE_HEIGHT

class CreateLMDB(object):
    def __init__(self, train, trLabel, val, valLabel):
            self.train = train
            self.trLabel = trLabel
            self.val = val
            self.valLabel = valLabel

            self.create_array_datum()
            self.create_train_lmdb()
            self.create_val_lmdb()
```

```
        def make_datum(self, img, label):
                return caffe_pb2.Datum(
                        channels=3,
                        width=IMAGE_WIDTH,
                        height=IMAGE_HEIGHT,
                        label=label,
                        data=np.rollaxis(img, 2).tostring())


        def create_train_lmdb(self):
                print('Creating train_lmdb')

                random.shuffle(self.trDatum)
                train_lmdb = settings.database_path + "train_lmdb"

                # remove if there are existing directories
                if os.path.isdir(settings.database_path + "train_lmdb"):
                        os.system("rmdir /s /q " + settings.database_path + "train_lmdb")
                        print("Existing TRAIN LMDB files were deleted.")

                in_db = lmdb.open(train_lmdb, map_size=int(1e12))
                with in_db.begin(write=True) as in_txn:
                        for in_idx, datum in enumerate(self.trDatum):
                                in_txn.put('{:0>5d}'.format(in_idx).encode(), datum.SerializeToString())
                in_db.close()

                print ('\nDone creating train_lmdb')

        # accepts an n x 2 array of mapping: filename-label
        def create_val_lmdb(self):

                print ('\nCreating validation_lmdb')

                random.shuffle(self.valDatum)
                val_lmdb = settings.database_path + "val_lmdb"

                # remove if there are existing directories
                if os.path.isdir(settings.database_path + "val_lmdb"):
                        os.system("rmdir /s /q " + settings.database_path + "val_lmdb")
                        print("Existing VAL LMDB files were deleted.")


                in_db = lmdb.open(val_lmdb, map_size=int(1e12))
                with in_db.begin(write=True) as in_txn:
                        for in_idx, datum in enumerate(self.valDatum):
                                in_txn.put('{:0>5d}'.format(in_idx).encode(), datum.SerializeToString())
                in_db.close()

                print ('\nDone creating val_lmdb')

        # accepts a numpy array of images and corresponding labels
        def create_array_datum(self):
                self.trDatum = []
                self.valDatum = []

                for index, img in enumerate(self.train):
                        datum = self.make_datum(img, self.trLabel[index])
                        self.trDatum.append(datum)

                for index, img in enumerate(self.val):
                        datum = self.make_datum(img, self.valLabel[index])
                        self.valDatum.append(datum)


import time
import sys
from PyQt5 import *
from PyQt5.QtWidgets import *
from PyQt5.QtWebKit import *
from PyQt5.QtWebKitWidgets import *
from PyQt5.QtCore import QUrl
from PyQt5.QtPrintSupport import QPrintDialog, QPrinter
from PyQt5.QtGui import QTextDocument

class createHTML(object):
        def __init__(self, filename, classes):
                self.filename = filename
                self.classes = classes

        def createHeader(self, model, thold):
                with open(self.filename, 'w') as myFile:
                        myFile.write('<html>')
                        myFile.write('<body>')
                        myFile.write('<h2>')
                        myFile.write('RaDSSv02 Classifier Application Results')
                        myFile.write('</h2>')
                        myFile.write('<h4>Model Used: ' + model + '</h4>')
                        myFile.write('<h4>Threshold: ' + thold + '</h4>')
                        myFile.write('<br/>')
                        myFile.write('<table style="width:100%"><tr><th>Image</th><th>Details</th></tr>')

        def createImg(self, img, probab, cls, details):
                with open(self.filename, 'a') as myFile:
                        myFile.write('<tr>')
                        myFile.write('<td><img src="' + img +
                        '" height="256"width="256" /> </td>')
                        myFile.write('<td><p><b>Name:</b> ' + img + '<br />')
                        myFile.write('<b>Class:</b> ' + cls + ' <br />')
                        myFile.write('<b>Probabilities:</b>')
                        myFile.write('<ul>')
                        for idx, clsname in enumerate(self.classes):
                                myFile.write('<li>' + clsname + ' : ' + str(probab[idx]) + '</li>')
                        myFile.write('</ul></p></tr>')
        def createEnd(self):
                with open(self.filename, 'a') as myFile:
                        myFile.write('</table>')
                        myFile.write('</body>')
                        myFile.write('</html>')

        def convertToPDF(self, outFile):
                doc = QTextDocument()
                location = self.filename
                html = open(location).read()
                doc.setHtml(html)

                printer = QPrinter()
                printer.setOutputFileName(outFile)
                printer.setOutputFormat(QPrinter.PdfFormat)
                printer.setPageSize(QPrinter.A4);
                printer.setPageMargins(3,3,3,3, QPrinter.Millimeter);
                doc.print_(printer)

        def save_pdf(self, widg):
                options = QFileDialog.Options()
                options |= QFileDialog.DontUseNativeDialog
                fileName, _ = QFileDialog.getSaveFileName(widg,
                "Export to PDF","","PDF Files (*.pdf)", options=options)
                if fileName:
                        self.convertToPDF(fileName)
                        return fileName


# Written by https://github.com/aleju/imgaug

from PIL import Image
import imgaug as ia
from imgaug import augmenters as iaa
import numpy as np
import cv2
import settings
import glob
import random

# random example images
# images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)

# Sometimes(0.5, ...) applies the given augmenter in 50% of all cases,
# e.g. Sometimes(0.5, GaussianBlur(0.3)) would blur roughly every second image.
sometimes = lambda aug: iaa.Sometimes(0.5, aug)

# Define our sequence of augmentation steps that will be applied to every image
# All augmenters with per_channel=0.5 will sample one value _per image_
# in 50% of all cases. In all other cases they will sample new values
# _per channel_.
seq = iaa.Sequential(
    [
        # apply the following augmenters to most images
        iaa.Fliplr(0.5), # horizontally flip 50% of all images
        iaa.Flipud(0.2), # vertically flip 20% of all images
        # crop images by -5% to 10% of their height/width
        sometimes(iaa.CropAndPad(
                percent=(-0.05, 0.1),
                pad_mode=ia.ALL,
                pad_cval=(0, 255)
        )),
        sometimes(iaa.Affine(
                scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
                        # scale images to 80-120% of their size, individually per axis
                translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
                        # translate by -20 to +20 percent (per axis)
                rotate=(-45, 45), # rotate by -45 to +45 degrees
                shear=(-16, 16), # shear by -16 to +16 degrees
                order=[0, 1], # use nearest neighbour or bilinear interpolation (fast)
                cval=(0, 255), # if mode is constant, use a cval between 0 and 255
                mode=ia.ALL
                        # use any of scikit-image's warping modes (see 2nd image from the top for examples)
        )),
        # execute 0 to 5 of the following (less important) augmenters per image
        # don't execute all of them, as that would often be way too strong
        iaa.SomeOf((0, 5),
                [
                sometimes(iaa.Superpixels(p_replace=(0, 1.0), n_segments=(20, 200))),
                                # convert images into their superpixel representation
                iaa.OneOf([
                        iaa.GaussianBlur((0, 3.0)),
                                    # blur images with a sigma between 0 and 3.0
                        iaa.AverageBlur(k=(2, 7)),
                                    # blur image using local means with kernel sizes between 2 and 7
                        iaa.MedianBlur(k=(3, 11)),
                                    # blur image using local medians with kernel sizes between 2 and 7
                    ]),
                    iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5)), # sharpen images
                    iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0)), # emboss images
                    # search either for all edges or for directed edges,
                    # blend the result with the original image using a blobby mask
                    iaa.SimplexNoiseAlpha(iaa.OneOf([
                        iaa.EdgeDetect(alpha=(0.5, 1.0)),
                        iaa.DirectedEdgeDetect(alpha=(0.5, 1.0), direction=(0.0, 1.0)),
                    ])),
                    iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05*255), per_channel=0.5),
                                # add gaussian noise to images
                    iaa.OneOf([
                        iaa.Dropout((0.01, 0.1), per_channel=0.5), # randomly remove up to 10% of the pixels
                        iaa.CoarseDropout((0.03, 0.15), size_percent=(0.02, 0.05), per_channel=0.2),
                    ]),
                    iaa.Invert(0.05, per_channel=True), # invert color channels
                    iaa.Add((-10, 10), per_channel=0.5),
                                # change brightness of images (by -10 to 10 of original value)
                    iaa.AddToHueAndSaturation((-20, 20)), # change hue and saturation
                    # either change the brightness of the whole image (sometimes
                    # per channel) or change the brightness of subareas
                    iaa.OneOf([
                        iaa.Multiply((0.5, 1.5), per_channel=0.5),
                        iaa.FrequencyNoiseAlpha(
                            exponent=(-4, 0),
                            first=iaa.Multiply((0.5, 1.5), per_channel=True),
                            second=iaa.ContrastNormalization((0.5, 2.0))
                        )
                    ]),
                    iaa.ContrastNormalization((0.5, 2.0), per_channel=0.5), # improve or worsen the contrast
                    iaa.Grayscale(alpha=(0.0, 1.0)),
                    sometimes(iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)),
                                # move pixels locally around (with random strengths)
                    sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05))),
                                # sometimes move parts of the image around
                    sometimes(iaa.PerspectiveTransform(scale=(0.01, 0.1)))
                ],
                random_order=True
            )
        ],
        random_order=True
)
# accepts an image array and returns a k number of augmented images
def augment(img, k):
        images_aug = []
        # since seq is random, we can perform same sequence different order k times
        for i in range(k):
                images_aug.append(seq.augment_image(img))

        return images_aug


from shutil import copy2, move
from readFiles import FileReader
import glob

dst = "C:\Thesis\CODES\DatasetCopied\TrainValP"
src = "C:\Thesis\CODES\DatasetCombined"
lab = "C:/Thesis/CODES/00DatasetLabelsss/TrainValP.xlsx"

#based on labels file but this is tuple with labels
reader = FileReader(lab)
file = reader.get_data_and_labels()
imageFiles = glob.glob(src + "/*.jpg")

for i in range(len(file)):
        halu = 0
        for imgpath in imageFiles:
                if(file[0][i] in imgpath):
                        copy2(imgpath, dst)
                        halu = 1
        if(halu == 0):
                print(file[0][i])

print("COPIED!")


import caffe
import lmdb
import numpy as np
import matplotlib.pyplot as plt
from caffe.proto import caffe_pb2
# Wei Yang 2015-08-19
# Source
#   Read LevelDB/LMDB
#   ==================
#       http://research.beenfrog.com/code/2015/03/28/read-leveldb-lmdb-for-caffe-with-python.html
#    Plot image
#    ==================
#         http://www.pyimagesearch.com/2014/11/03/display-matplotlib-rgb-image/
#    Creating LMDB in python
#    ==================
#         http://deepdish.io/2015/04/28/creating-lmdb-in-python/


def checkdb(lmdb_file):
        lmdb_env = lmdb.open(lmdb_file)
        lmdb_txn = lmdb_env.begin()
        lmdb_cursor = lmdb_txn.cursor()
        datum = caffe_pb2.Datum()

        i = 0
        for key, value in lmdb_cursor:
                datum.ParseFromString(value)

                label = datum.label
                data = caffe.io.datum_to_array(datum)
                im = data.astype(np.uint8)
                im = np.transpose(im, (2, 1, 0)) # original (dim, col, row)
                print(i, " label ", label)
                i += 1
                plt.imshow(im)
                plt.show()

lmdb_file = "C:/Thesis/CODES/RaDSSv02/database/train_lmdb"
checkdb(lmdb_file)

lmdb_file = "C:/Thesis/CODES/RaDSSv02/database/val_lmdb"
checkdb(lmdb_file)


#!/usr/bin/env python
import datetime
import os
import sys

def extract_datetime_from_line(line, year):
    # Expected format: I0210 13:39:22.381027 25210 solver.cpp:204] Iteration 100, lr = 0.00992565
    line = line.strip().split()
    month = int(line[0][1:3])
    day = int(line[0][3:])
    timestamp = line[1]
    pos = timestamp.rfind('.')
    ts = [int(x) for x in timestamp[:pos].split(':')]
    hour = ts[0]
    minute = ts[1]
    second = ts[2]
    microsecond = int(timestamp[pos + 1:])
    dt = datetime.datetime(year, month, day, hour, minute, second, microsecond)
    return dt


def get_log_created_year(input_file):
    """Get year from log file system timestamp
    """

    log_created_time = os.path.getctime(input_file)
    log_created_year = datetime.datetime.fromtimestamp(log_created_time).year
    return log_created_year


def get_start_time(line_iterable, year):
    """Find start time from group of lines
    """

    start_datetime = None
    for line in line_iterable:
        line = line.strip()
        if line.find('Solving') != -1:
            start_datetime = extract_datetime_from_line(line, year)
            break
    return start_datetime


def extract_seconds(input_file, output_file):
    with open(input_file, 'r') as f:
        lines = f.readlines()
    log_created_year = get_log_created_year(input_file)
    start_datetime = get_start_time(lines, log_created_year)
    assert start_datetime, 'Start time not found'

    last_dt = start_datetime
    out = open(output_file, 'w')
    for line in lines:
        line = line.strip()
        if line.find('Iteration') != -1:
            dt = extract_datetime_from_line(line, log_created_year)

            # if it's another year
            if dt.month < last_dt.month:
                log_created_year += 1
                dt = extract_datetime_from_line(line, log_created_year)
            last_dt = dt

            elapsed_seconds = (dt - start_datetime).total_seconds()
            out.write('%f\n' % elapsed_seconds)
    out.close()

if __name__ == '__main__':
    if len(sys.argv) < 3:
        print('Usage: ./extract_seconds input_file output_file')
        exit(1)
    extract_seconds(sys.argv[1], sys.argv[2])


#!/usr/bin/env python

"""
Parse training log

Evolved from parse_log.sh
"""

import os
import re
import extract_seconds
import argparse
import csv
from collections import OrderedDict


def parse_log(path_to_log):
    """Parse log file
    Returns (train_dict_list, test_dict_list)

    train_dict_list and test_dict_list are lists of dicts that define the table
    rows
    """

    regex_iteration = re.compile('Iteration (\d+)')
    regex_train_output = re.compile('Train net output #(\d+): (\S+) = ([\.\deE+-]+)')
    regex_test_output = re.compile('Test net output #(\d+): (\S+) = ([\.\deE+-]+)')
    regex_learning_rate = re.compile('lr = ([-+]?[0-9]*\.?[0-9]+([eE]?[-+]?[0-9]+)?)')

    # Pick out lines of interest
    iteration = -1
    learning_rate = float('NaN')
    train_dict_list = []
    test_dict_list = []
    train_row = None
    test_row = None

    logfile_year = extract_seconds.get_log_created_year(path_to_log)
    with open(path_to_log) as f:
        start_time = extract_seconds.get_start_time(f, logfile_year)
        last_time = start_time

        for line in f:
            iteration_match = regex_iteration.search(line)
            if iteration_match:
                iteration = float(iteration_match.group(1))
            if iteration == -1:
                # Only start parsing for other stuff if we've found the first
                # iteration
                continue

            try:
                time = extract_seconds.extract_datetime_from_line(line,
                                                                  logfile_year)
            except ValueError:
                # Skip lines with bad formatting, for example when resuming solver
                continue

            # if it's another year
            if time.month < last_time.month:
                logfile_year += 1
                time = extract_seconds.extract_datetime_from_line(line, logfile_year)
            last_time = time

            seconds = (time - start_time).total_seconds()

            learning_rate_match = regex_learning_rate.search(line)
            if learning_rate_match:
                learning_rate = float(learning_rate_match.group(1))

            train_dict_list, train_row = parse_line_for_net_output(
                regex_train_output, train_row, train_dict_list,
                line, iteration, seconds, learning_rate
            )
            test_dict_list, test_row = parse_line_for_net_output(
                regex_test_output, test_row, test_dict_list,
                line, iteration, seconds, learning_rate
            )

    fix_initial_nan_learning_rate(train_dict_list)
    fix_initial_nan_learning_rate(test_dict_list)

    return train_dict_list, test_dict_list


def parse_line_for_net_output(regex_obj, row, row_dict_list,
                              line, iteration, seconds, learning_rate):
    """Parse a single line for training or test output

    Returns a a tuple with (row_dict_list, row)
    row: may be either a new row or an augmented version of the current row
    row_dict_list: may be either the current row_dict_list or an augmented
    version of the current row_dict_list
    """

    output_match = regex_obj.search(line)
    if output_match:
        if not row or row['NumIters'] != iteration:
            # Push the last row and start a new one
            if row:
                # If we're on a new iteration, push the last row
                # This will probably only happen for the first row; otherwise
                # the full row checking logic below will push and clear full
                # rows
                row_dict_list.append(row)

            row = OrderedDict([
                ('NumIters', iteration),
                ('Seconds', seconds),
                ('LearningRate', learning_rate)
            ])

        # output_num is not used; may be used in the future
        # output_num = output_match.group(1)
        output_name = output_match.group(2)
        output_val = output_match.group(3)
        row[output_name] = float(output_val)

    if row and len(row_dict_list) >= 1 and len(row) == len(row_dict_list[0]):
        # The row is full, based on the fact that it has the same number of
        # columns as the first row; append it to the list
        row_dict_list.append(row)
        row = None

    return row_dict_list, row


def fix_initial_nan_learning_rate(dict_list):
    """Correct initial value of learning rate

    Learning rate is normally not printed until after the initial test and
    training step, which means the initial testing and training rows have
    LearningRate = NaN. Fix this by copying over the LearningRate from the
    second row, if it exists.
    """

    if len(dict_list) > 1:
        dict_list[0]['LearningRate'] = dict_list[1]['LearningRate']


def save_csv_files(logfile_path, output_dir, train_dict_list, test_dict_list,
                   delimiter=',', verbose=False):
    """Save CSV files to output_dir

    If the input log file is, e.g., caffe.INFO, the names will be
    caffe.INFO.train and caffe.INFO.test
    """

    log_basename = os.path.basename(logfile_path)
    train_filename = os.path.join(output_dir, log_basename + '.train')
    write_csv(train_filename, train_dict_list, delimiter, verbose)

    test_filename = os.path.join(output_dir, log_basename + '.test')
    write_csv(test_filename, test_dict_list, delimiter, verbose)


def write_csv(output_filename, dict_list, delimiter, verbose=False):
    """Write a CSV file
    """

    if not dict_list:
        if verbose:
            print('Not writing %s; no lines to write' % output_filename)
        return

    dialect = csv.excel
    dialect.delimiter = delimiter
    with open(output_filename, 'w') as f:
        dict_writer = csv.DictWriter(f, fieldnames=dict_list[0].keys(),
                                     dialect=dialect)
        dict_writer.writeheader()
        dict_writer.writerows(dict_list)
    if verbose:
        print('Wrote %s' % output_filename)


def parse_args():
    description = ('Parse a Caffe training log into two CSV files '
                   'containing training and testing information')
    parser = argparse.ArgumentParser(description=description)

    parser.add_argument('logfile_path',
                        help='Path to log file')

    parser.add_argument('output_dir',
                        help='Directory in which to place output CSV files')

    parser.add_argument('--verbose',
                        action='store_true',
                        help='Print some extra info (e.g., output filenames)')

    parser.add_argument('--delimiter',
                        default=',',
                        help=('Column delimiter in output files '
                              '(default: \'%(default)s\')'))

    args = parser.parse_args()
    return args

if __name__ == '__main__':
    args = parse_args()
    train_dict_list, test_dict_list = parse_log(args.logfile_path)
    save_csv_files(args.logfile_path, args.output_dir, train_dict_list,
                   test_dict_list, delimiter=args.delimiter, verbose=args.verbose)

'''
Title           :plot_learning_curve.py
Description     :This script generates learning curves for caffe models
Author          :Adil Moujahid
Date Created    :20160619
Date Modified   :20160619
version         :0.1
usage           :python plot_learning_curve.py model_1_train.log
                            ./caffe_model_1_learning_curve.png
python_version  :2.7.11
'''

import os
import sys
import subprocess
import pandas as pd
import parse_log
import settings
import matplotlib
#matplotlib.use('Agg')
import matplotlib.pylab as plt
plt.style.use('ggplot')

def plot_curves(log_path, output_path, plot_path):

        '''
        Parse log file using parse_log.py
        '''
        os.system("python parse_log.py " + log_path + " " + output_path)

        #Read training and test logs
        train_log_path = log_path + '.train'
        test_log_path = log_path + '.test'
        train_log = pd.read_csv(train_log_path, header=[0])
        test_log = pd.read_csv(test_log_path, header=[0])


        '''
        Making learning curve
        '''
        fig, ax1 = plt.subplots()

        #Plotting training and test losses
        train_loss, = ax1.plot(train_log['NumIters'], train_log['loss'], color='red',  alpha=.5)
        test_loss, = ax1.plot(test_log['NumIters'], test_log['loss'], linewidth=2, color='green')
        ax1.set_ylim(ymin=0, ymax=1)
        ax1.set_xlabel('Iterations', fontsize=12)
        ax1.set_ylabel('Loss', fontsize=12)
        ax1.tick_params(labelsize=12)
        #Plotting test accuracy
        ax2 = ax1.twinx()
        test_accuracy, = ax2.plot(test_log['NumIters'], test_log['accuracy'],
        linewidth=2, color='blue')
        ax2.set_ylim(ymin=0, ymax=1)
        ax2.set_ylabel('Accuracy', fontsize=12)
        ax2.tick_params(labelsize=12)
        #Adding legend
        plt.legend([train_loss, test_loss, test_accuracy], ['Training Loss',
        'Test Loss', 'Test Accuracy'],  bbox_to_anchor=(1, 0.2))
        plt.title('Training Curve', fontsize=15)
        #Saving learning curve
        plt.savefig(plot_path)

        '''
        Deleting training and test logs
        '''
        command = 'bash -c "rm ' + train_log_path + "\""
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE)
        process.wait()

        command = 'bash -c "rm ' + test_log_path + "\""
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE)
        process.wait()

        '''
        Get the last accuracy obtained by the model
        '''
        print(test_log['accuracy'].iloc[-1])
        return test_log['accuracy'].iloc[-1]


import inspect
import os
import random
import sys
import matplotlib.cm as cmx
import matplotlib.colors as colors
import matplotlib.pyplot as plt
import matplotlib.legend as lgd
import matplotlib.markers as mks

def get_log_parsing_script():
    dirname = os.path.dirname(os.path.abspath(inspect.getfile(
        inspect.currentframe())))
    return dirname + '/parse_log.sh'

def get_log_file_suffix():
    return '.log'

def get_chart_type_description_separator():
    return '  vs. '

def is_x_axis_field(field):
    x_axis_fields = ['Iters', 'Seconds']
    return field in x_axis_fields

def create_field_index():
    train_key = 'Train'
    test_key = 'Test'
    field_index = {train_key:{'Iters':0, 'Seconds':1, train_key + ' loss':2,
                              train_key + ' learning rate':3},
                   test_key:{'Iters':0, 'Seconds':1, test_key + ' accuracy':2,
                             test_key + ' loss':3}}
    fields = set()
    for data_file_type in field_index.keys():
        fields = fields.union(set(field_index[data_file_type].keys()))
    fields = list(fields)
    fields.sort()
    return field_index, fields

def get_supported_chart_types():
    field_index, fields = create_field_index()
    num_fields = len(fields)
    supported_chart_types = []
    for i in range(num_fields):
        if not is_x_axis_field(fields[i]):
            for j in range(num_fields):
                if i != j and is_x_axis_field(fields[j]):
                    supported_chart_types.append('%s%s%s' % (
                        fields[i], get_chart_type_description_separator(),
                        fields[j]))
    return supported_chart_types

def get_chart_type_description(chart_type):
    supported_chart_types = get_supported_chart_types()
    chart_type_description = supported_chart_types[chart_type]
    return chart_type_description

def get_data_file_type(chart_type):
    description = get_chart_type_description(chart_type)
    data_file_type = description.split()[0]
    return data_file_type

def get_data_file(chart_type, path_to_log):
    return (os.path.basename(path_to_log) + '.' +
            get_data_file_type(chart_type).lower())

def get_field_descriptions(chart_type):
    description = get_chart_type_description(chart_type).split(
        get_chart_type_description_separator())
    y_axis_field = description[0]
    x_axis_field = description[1]
    return x_axis_field, y_axis_field

def get_field_indices(x_axis_field, y_axis_field):
    data_file_type = get_data_file_type(chart_type)
    fields = create_field_index()[0][data_file_type]
    return fields[x_axis_field], fields[y_axis_field]

def load_data(data_file, field_idx0, field_idx1):
        data = [[], []]
        with open(data_file, 'r') as f:
                for line in f:
                        line = line.strip()
                        if line[0] != '#':
                                print(field_idx0)
                                print(field_idx1)
                                fields = line.split()
                                data[0].append(float(fields[field_idx0].strip()))
                                data[1].append(float(fields[field_idx1].strip()))
        return data

def random_marker():
    markers = mks.MarkerStyle.markers
    num = len(markers.keys())
    idx = random.randint(0, num - 1)
    return list(markers.keys())[idx]

def get_data_label(path_to_log):
    label = path_to_log[path_to_log.rfind('/')+1 : path_to_log.rfind(
        get_log_file_suffix())]
    return label

def get_legend_loc(chart_type):
    x_axis, y_axis = get_field_descriptions(chart_type)
    loc = 'lower right'
    if y_axis.find('accuracy') != -1:
        pass
    if y_axis.find('loss') != -1 or y_axis.find('learning rate') != -1:
        loc = 'upper right'
    return loc

def plot_chart(chart_type, path_to_png, path_to_log_list):
    for path_to_log in path_to_log_list:
        os.system('%s %s' % (get_log_parsing_script(), path_to_log))
        data_file = get_data_file(chart_type, path_to_log)
        x_axis_field, y_axis_field = get_field_descriptions(chart_type)
        x, y = get_field_indices(x_axis_field, y_axis_field)
        data = load_data(data_file, x, y)
        ## TODO: more systematic color cycle for lines
        color = [random.random(), random.random(), random.random()]
        label = get_data_label(path_to_log)
        linewidth = 0.75
        ## If there too many datapoints, do not use marker.
##        use_marker = False
        use_marker = True
        if not use_marker:
            plt.plot(data[0], data[1], label = label, color = color,
                     linewidth = linewidth)
        else:
            marker = random_marker()
            plt.plot(data[0], data[1], label = label, color = color,
                     marker = marker, linewidth = linewidth)
    legend_loc = get_legend_loc(chart_type)
    plt.legend(loc = legend_loc, ncol = 1) # ajust ncol to fit the space
    plt.title(get_chart_type_description(chart_type))
    plt.xlabel(x_axis_field)
    plt.ylabel(y_axis_field)
    plt.savefig(path_to_png)
    plt.show()

def print_help():
        print("""This script mainly serves as the basis of your customizations.
        Customization is a must.
        You can copy, paste, edit them in whatever way you want.
        Be warned that the fields in the training log may change in the future.
        You had better check the data files and change the mapping from field name to
        field index in create_field_index before designing your own plots.
        Usage:
        ./plot_training_log.py chart_type[0-%s] /where/to/save.png /path/to/first.log ...
        Notes:
        1. Supporting multiple logs.
        2. Log file name must end with the lower-cased "%s".
        Supported chart types:""" % (len(get_supported_chart_types()) - 1, get_log_file_suffix()))
        supported_chart_types = get_supported_chart_types()
        num = len(supported_chart_types)
        for i in range(num):
                print ('    %d: %s' % (i, supported_chart_types[i]))
        sys.exit()

def is_valid_chart_type(chart_type):
    return chart_type >= 0 and chart_type < len(get_supported_chart_types())

if __name__ == '__main__':
    if len(sys.argv) < 4:
        print_help()
    else:
        chart_type = int(sys.argv[1])
        if not is_valid_chart_type(chart_type):
            print ('%s is not a valid chart type.' % chart_type)
            print_help()
        path_to_png = sys.argv[2]
        if not path_to_png.endswith('.png'):
            print ('Path must ends with png' % path_to_png)
            sys.exit()
        path_to_logs = sys.argv[3:]
        for path_to_log in path_to_logs:
            if not os.path.exists(path_to_log):
                print ('Path does not exist: %s' % path_to_log)
                sys.exit()
            if not path_to_log.endswith(get_log_file_suffix()):
                print ('Log file must end in %s.' % get_log_file_suffix())
                print_help()
        ## plot_chart accpets multiple path_to_logs
        plot_chart(chart_type, path_to_png, path_to_logs)


import cv2
import settings
import numpy as np

class PreprocessImage(object):
        def __init__(self):
                pass

        def transform_img(img, img_width=settings.IMAGE_WIDTH, img_height=settings.IMAGE_HEIGHT):

                #Histogram Equalization
                img[:, :, 0] = cv2.equalizeHist(img[:, :, 0])
                img[:, :, 1] = cv2.equalizeHist(img[:, :, 1])
                img[:, :, 2] = cv2.equalizeHist(img[:, :, 2])

                # need not to do this because caffe accepts BGR
                #img = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

                #Image Resizing
                img = cv2.resize(img, (img_width, img_height), interpolation = cv2.INTER_CUBIC)

                return img

        # resize and pad an image and keep aspect ratio
        def resizeAndPad(img, size = (settings.IMAGE_WIDTH, settings.IMAGE_HEIGHT), padColor=0):
                h, w = img.shape[:2]
                sh, sw = size

                # interpolation method
                if h > sh or w > sw: # shrinking image
                        interp = cv2.INTER_AREA
                else: # stretching image
                        interp = cv2.INTER_CUBIC

                # aspect ratio of image
                aspect = w/h  # if on Python 2, you might need to cast as a float: float(w)/h

                # compute scaling and pad sizing
                if aspect > 1: # horizontal image
                        new_w = sw
                        new_h = np.round(new_w/aspect).astype(int)
                        pad_vert = (sh-new_h)/2
                        pad_top, pad_bot = np.floor(pad_vert).astype(int), np.ceil(pad_vert).astype(int)
                        pad_left, pad_right = 0, 0
                elif aspect < 1: # vertical image
                        new_h = sh
                        new_w = np.round(new_h*aspect).astype(int)
                        pad_horz = (sw-new_w)/2
                        pad_left, pad_right = np.floor(pad_horz).astype(int), np.ceil(pad_horz).astype(int)
                        pad_top, pad_bot = 0, 0
                else: # square image
                        new_h, new_w = sh, sw
                        pad_left, pad_right, pad_top, pad_bot = 0, 0, 0, 0

                # set pad color
                if len(img.shape) is 3 and not isinstance(padColor, (list, tuple, np.ndarray)):
                # color image but only one color provided
                        padColor = [padColor]*3

                # scale and pad
                scaled_img = cv2.resize(img, (new_w, new_h), interpolation=interp)
                scaled_img = cv2.copyMakeBorder(scaled_img, pad_top, pad_bot, pad_left, pad_right,
                borderType=cv2.BORDER_CONSTANT, value=padColor)

                return scaled_img


import pandas
import re
import settings
import datetime
import google.protobuf.text_format as txtf
import numpy as np
from caffe.proto import caffe_pb2
from pandas import ExcelWriter
from pandas import ExcelFile
from PyQt5.QtWidgets import *

class FileReader(object):
        def __init__(self, filename):
                self.filename = filename
                self.data = []

                # only excel files (.xls) are accpeted
                if bool(re.search('.xls', self.filename)):
                        self.open_excel_file()

        def open_excel_file(self):
                print("Open Excel File")
                self.data = pandas.read_excel(self.filename, header=None)

        def get_data_and_labels(self):
                # will have to edit this to be a list
                # synatx will be self.data.values.tolist()
                # self.data.values is a numpy array
                return self.data

        def get_image_label(self, image_filename):
                for data in self.data:
                        if(data[0] == image_filename):          # if matched
                                return data[1]
                return None

        def export_to_excel(self, widg, data):
                arr = np.array(data)
                df = pandas.DataFrame({'Absolute Paths': arr[:,0],
                                        'Labels': arr[:,1]})
                options = QFileDialog.Options()
                options |= QFileDialog.DontUseNativeDialog
                fileName, _ = QFileDialog.getSaveFileName(widg, "Save Excel File","",
                "Excel Files (*.xls *.xlsx)", options=options)
                if fileName:
                        writer = ExcelWriter(fileName)
                        df.to_excel(writer,'Sheet1',index=False, header=False)
                        writer.save()
                        return fileName

        def edit_train_prototxt(self, modelname, bs, output):
                net = caffe_pb2.NetParameter()
                filename = settings.models_path + modelname + "/train_val.prototxt"

                fn = filename
                with open(fn) as f:
                        s = f.read()
                        txtf.Merge(s, net)

                layerNames = [l.name for l in net.layer]
                idx = layerNames.index('data')

                # first data layer
                l = net.layer[idx]
                l.transform_param.mean_file = settings.data_path + "images_mean.binaryproto"
                l.transform_param.crop_size = settings.IMAGE_WIDTH
                l.data_param.source = settings.database_path + "train_lmdb"
                l.data_param.batch_size = bs

                # second data layer
                l = net.layer[idx+1]
                l.transform_param.mean_file = settings.data_path + "images_mean.binaryproto"
                l.transform_param.crop_size = settings.IMAGE_WIDTH
                l.data_param.source = settings.database_path + "val_lmdb"
                # edit pa ba to of deafult to the model na?
                #l.data_param.batch_size = settings.val_bs

                # edit number of classes based on the data
                idx = layerNames.index('fc8')
                l = net.layer[idx]
                l.inner_product_param.num_output = output

                # write it on the same file
                outFn = filename
                print('writing', outFn)
                with open(outFn, 'w') as f:
                        f.write(str(net))
```

```
def solver(self, modelname, base_lr, stepsize, max_iter, mode):
        s = caffe_pb2.SolverParameter()

        # Specify locations of the train and (maybe) test networks.
        s.net = settings.models_path + modelname + "/train_val.prototxt"

        s.test_interval = 1000  # Test after every 1000 training iterations.
        s.test_iter.append(100)         # Test on 100 batches each time we test.

        # The number of iterations over which to average the gradient.
        # Effectively boosts the training batch size by the given factor, without
        # affecting memory utilization.
        #s.iter_size = 1

        s.max_iter = max_iter    # of times to update the net (training iterations)

        # Solve using the stochastic gradient descent (SGD) algorithm.
        # Other choices include 'Adam' and 'RMSProp'.
        #s.type = 'SGD'

        # Set the initial learning rate for SGD.
        s.base_lr = base_lr

        # Set `lr_policy` to define how the learning rate changes during training.
        # Here, we 'step' the learning rate by multiplying it by a factor `gamma`
        # every `stepsize` iterations.
        s.lr_policy = 'step'
        s.gamma = 0.1
        s.stepsize = stepsize

        # Set other SGD hyperparameters. Setting a non-zero `momentum` takes a
        # weighted average of the current gradient and previous gradients to make
        # learning more stable. L2 weight decay regularizes learning, to help prevent
        # the model from overfitting.
        s.momentum = 0.9
        s.weight_decay = 5e-4

        # Display the current training loss and accuracy every 1000 iterations.
        s.display = 1000

        # Snapshots are files used to store networks we've trained.  Here, we'll
        # snapshot every 10K iterations -- ten times during training.
        s.snapshot = 10000
        now = datetime.datetime.now().strftime("%Y%m%d_%H%M")
        s.snapshot_prefix = settings.models_path + modelname + "_" + str(now) + "_"

        # Train on the GPU.  Using the CPU to train large networks is very slow.
        s.solver_mode = caffe_pb2.SolverParameter.GPU
        if mode == 0:
                s.solver_mode = caffe_pb2.SolverParameter.CPU

        outFn = settings.models_path + modelname + "/solver.prototxt"
        print('Writing', outFn)
        with open(outFn, 'w') as f:
                f.write(str(s))

        # return caffemodel name thru snapshot name
        return modelname + "_" + str(now) + "_"

# edits:
# input_param.shape, num_output
def deploy(self, modelname, output):
        net = caffe_pb2.NetParameter()
        filename = settings.models_path + modelname + "/deploy.prototxt"

        fn = filename
        with open(fn) as f:
                s = f.read()
                txtf.Merge(s, net)

        layerNames = [l.name for l in net.layer]
        idx = layerNames.index('fc8')

        # edit number of classes based on the data
        l = net.layer[idx]
        l.inner_product_param.num_output = output

        # edit dimensionsx
        idx = layerNames.index('data')
        l = net.layer[idx]
        # remove existing shape
        x = l.input_param.shape.pop()
        # add the specifics based on settings
        x = l.input_param.shape.add()

        # assigns the values in settings
        # test batch size, channels, width, height
        x.dim[:] = [settings.test_bs, 3, settings.IMAGE_WIDTH, settings.IMAGE_HEIGHT]


        # write it on the same file
```

```
                outFn = filename
                print('writing', outFn)
                with open(outFn, 'w') as f:
                        f.write(str(net))

        def write_to_text(self, lst ,outFile):
                with open(outFile, 'w') as f:
                        for data in lst:
                                f.write(data[0] + " " + data[1])


# settings.py
# Setting Global Variables

#declarations caffe paths
global caffe_tools
global caffe_scripts

# declarations myapp
global ui_path
global train_path
global train_path_file
global test_path
global models_path
global database_path
global data_path
global tools_path
global help_path

# this is where to save the output files
# .caffemodels
# test_output.npy for predictions
# exported PDF

global save_files_path

#
global IMAGE_WIDTH
global IMAGE_HEIGHT
global icon_path
global val_bs
global test_bs

# model globals
global caffenet
global alexnet
global vggnet

# Classifier Model
global classifier_model

# initialize paths
# make sure that / is used
caffe_tools = "C:/Thesis/caffe/build/tools/Release"
caffe_scripts = "C:/Thesis/CODES/RaDSSv02/scripts/"

ui_path = "C:/Thesis/CODES/RaDSSv02/ui/"
models_path = "C:/Thesis/CODES/RaDSSv02/models/"

data_path = "C:/Thesis/CODES/RaDSSv02/data/"
tools_path = "C:/Thesis/caffe/build/tools/Release/"

help_path = "C:/Thesis/CODES/RaDSSv02/help/"

# invalid switch error will have to use \
database_path = "C:\Thesis\CODES\RaDSSv02\database\\"

# initialize image variables
IMAGE_WIDTH = 256
IMAGE_HEIGHT = 256
icon_path = "C:/Thesis/CODES/RaDSSv02/images/icon3.jpg"

# initialize batch sizes for val and test
val_bs = 100
test_bs = 10

#to be set by user
train_path = ""
train_path_file = ""
test_path = ""
save_files_path = ""

# will COMMENT if config show
train_path = "C:/Thesis/CODES/RaDSSv02/input/train/"
train_path_file = "C:/Thesis/CODES/RaDSSv02/input/train-label.xls"
test_path = "C:/Thesis/CODES/RaDSSv02/input/test/"
save_files_path = data_path

# model's values
# [lr, gamma, mom]

# caffenet values
```

```
caffenet = [0.01, 0.1, 0.9]

# alexnet values
alexnet = [0.01, 0.1, 0.9]

# vggnet values
vggnet = [0.0005, 0.001, 0.9]

# set classifier_model by the training administrator
# can be found in the models_path
classifier_model = "caffenet_20180523_071601_iter_1000.caffemodel"
#classifier_model = ""
```

XI. Acknowledgement

All glory and honor is to the Lord. I could never have survived this University if it weren't for His everlasting and unconditional love, and provisions of blessings. He was also the one who sent me great people to help me in this journey. I would like to thank my family, especially my mother for her support, and my sisters for their motivations. To Professor Marquez, who allowed me to use his collection of Radiolarians to increase my database, thank you sir. And to my adviser, Professor Solano, you are a blessing to me, po. Thank you for understanding and for your eternal support.