Deep Learning Application Engine (DLAE)

User Manual v1.0

Date: September 24, 2019

Table of Contents

1 Getting started

1.1 Requirements

Although it is possible to train and make predictions on deep learning (DL) models on a CPU, it is strongly suggested that users use a computer with a CUDA enabled GPU. General purpose GPUs can significantly reduce the amount of time required to develop DL models. A list of CUDA enable GPUs can be found here https://developer.nvidia.com/cuda-gpus.

Users will also need curated datasets to develop their DL models. The required format of the data is explained in a later section. There are a number of open-source datasets available on the internet. The choice of dataset depends on the area of research the user is involved in. Users are also encouraged to curate their own datasets from their research data. Although some public medical imaging datasets are available, they are not as easily acquired as some other datasets because of the sensitive nature of the information content contained within them. As such, users may need to curate their own medical imaging datasets to use with DLAE.

Python 3 (v3.6.8) is required to run the software. It can be downloaded from https://www.python.org/downloads/. It may be possible to use newer versions of Python as long as the version is supported by TensorFlow. However, I cannot guarantee that users won't run into issues with newer versions.

While not required, it is recommended that users also use an IDE. A popular and well maintained IDE, PyCharm, can be downloaded here https://www.jetbrains.com/pycharm/download/.

1.2 Installation

TensorFlow (TF) (v1.13.1) must be installed on the system. The GPU version of TensorFlow has its own set of prerequisites before it can be used. Detailed instruction can be found at https://www.tensorflow.org/install. I will briefly summarize what's required here (these are the steps for Windows; the process is similar for Linux systems):
1. Install the graphics driver for your CUDA enabled GPU. Find and download the required driver from https://www.nvidia.com/Download/index.aspx.
2. Install CUDA Toolkit 10.0 from https://developer.nvidia.com/cuda-downloads (you may need to click "Legacy Releases").
3. If you don't already have one, sign up for an NVIDIA developer account. Then login and download cuDNN 7.6.0 from https://developer.nvidia.com/cudnn. Extract the contents of the cuda .zip file then merge the sub-folders with the folder C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0 on your system.
4. Add the folder C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0 to your system's path variables.

You can now install TensorFlow v1.13.1. A number of other Python modules are required to run DLAE. The list of required modules are:

```
tensorflow-gpu==1.13.1
keras==2.2.4
imageio==2.5.0
opencv==3.4.2
keras-applications==1.0.7
scikit-learn==0.21.2
```

```
pillow==6.1.0
```

Alternatively, DLAE can also be installed using anaconda3, which can be downloaded here https://www.anaconda.com/distribution/. After installing anaconda3, simply run these three commands:

```
conda create -n dlae python==3.6.8
conda activate dlae
conda install tensorflow-gpu==1.13.1 keras==2.2.4 imageio==2.5.0
opencv==3.4.2 keras-applications==1.0.7 scikit-learn==0.21.2
pillow==6.1.0
```

Finally, clone the DLAE github repository from https://github.com/jeremiahws/dlae. If you don't have Git installed, you can download it from https://git-scm.com/downloads. To clone the repository, open a command prompt and run:

```
git clone https://github.com/jeremiahws/dlae.git
```

1.3 Application launching

There are three ways to interface with DLAE. The first way (GUI mode) is by utilizing the GUI to create a DLAE configuration structure to be processed by the engine. To launch the GUI, navigate to the top level directory of the DLAE repository and run:

```
python dlae.py
```

If successful, you will see the GUI displayed (Figure 1).

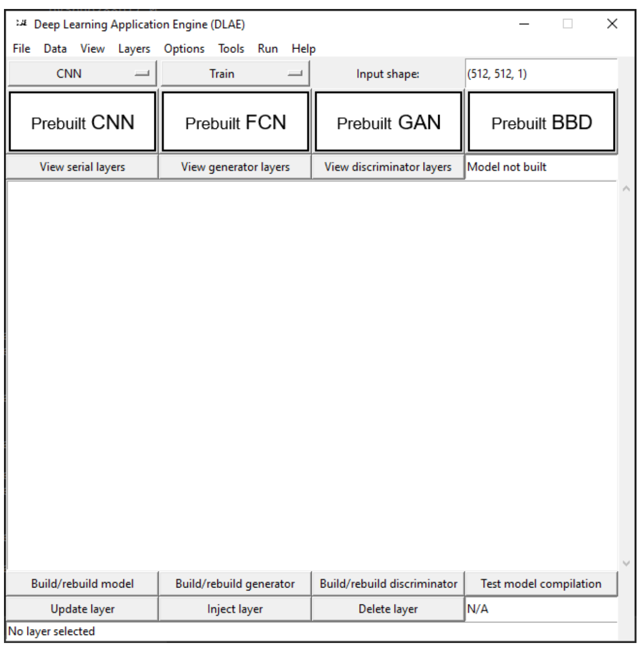

Figure 1. The DLAE GUI.

It's also possible to run DLAE in silent mode, whereby the GUI is suppressed. This requires the user to have already created a DLAE configuration file that they wish to process. To run DLAE in silent mode, run DLAE from the command line and specific the path to the configuration file as an input:

```
python dlae.py configuration_file_directory/config_file.json
```

The third way to run DLAE is by constructing an experiment generator that spawns DLAE configuration files. Each configuration file spawn form the generator would be a specific experiment that the user wishes to run on a given dataset. The DLAE engine then processes each configuration file spawn sequentially. This functionality is useful for ablation studies and hyperparameter searches/tuning. Experiment generators are covered in more detail in a later section.

2 Data curation

Data should be curated according to the type of DL model being developed and the specific task the user expects the DL model they are developing to perform.

2.1 Convolutional neural networks

For classification and regression tasks with convolutional neural networks (CNNs), the input data will be the images and the annotation will be a class label for that image. For 2D images, the images should have dimensions (n_{rows}, n_{cols}, n_{chans}), where n_{rows} is the number of rows, n_{cols} is the number of columns, and n_{chans} is the number of channels. For 3D images, the images should have dimensions (n_{rows}, n_{cols}, n_{slices}, n_{chans}), where n_{slices} is the number of slices.

For classification tasks, the annotation for an image will be a single integer corresponding to the class that the image belongs to. All class numbering schemes should start at 0 and increase linearly with an increment of 1. For example, for a dataset containing 3 image classes, the class labels should be in the range [0, 1, 2]. An example is shown in Figure 2.

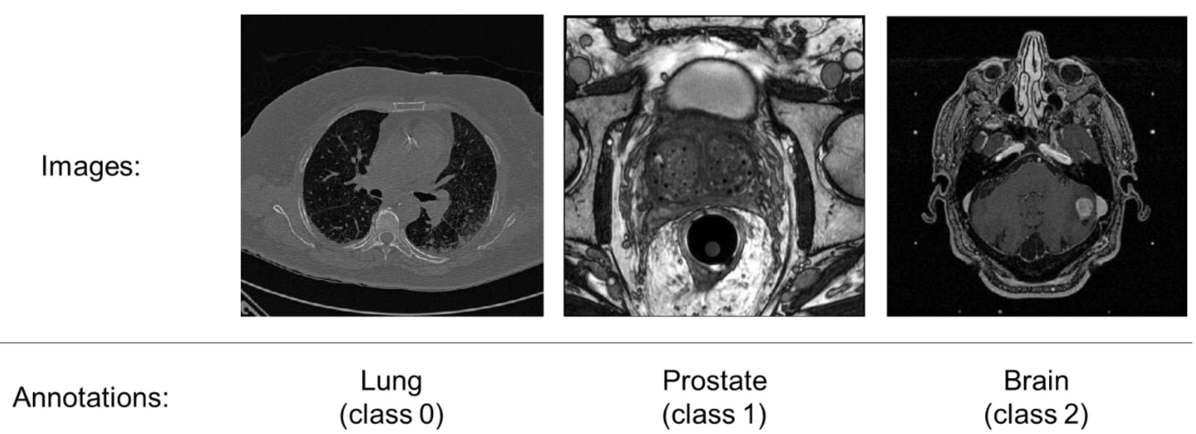

Figure 2. Example set of image/annotation pairs for an image classification problem with CNNs.

For regression tasks, the annotation for an image will be a row vector of length n_{coords} of numbers, where n_{coords} represents the number of coordinates to be regressed over. For example, if the centroid of the object in an image is at (row, col) = (234, 327), then the annotation would be [234, 327]. It should be noted that the coordinates can be floating point numbers.

2.2 Fully convolutional networks

For classification and regression tasks with fully convolutional networks (FCNs), the input data will be an image and the annotations will an array of pixel-wise labels for that image. For 2D images, both the images and annotations should have dimensions (n_{rows}, n_{cols}, $n_{chans1,2}$), where $n_{chans1,2}$ indicates that the number of channels for the image and annotation don't necessarily have to be equal (although the number of channels should be equal to 1 for pixel-wise classification problems). For 3D images, both the images and annotations should have dimensions (n_{rows}, n_{cols}, n_{slices}, $n_{chans1,2}$).

For pixel-wise classification tasks, the annotation for an image will be an array of integers corresponding to the classes of objects contained within each pixel of the image. All class numbering schemes should start at 0 and increase linearly with an increment of 1. For example, for a prostate image with 5 segmented organs (+1 class for background), the class labels should be in the range [0, 1, 2, 3, 4, 5, 6]. An example is shown in Figure 3.

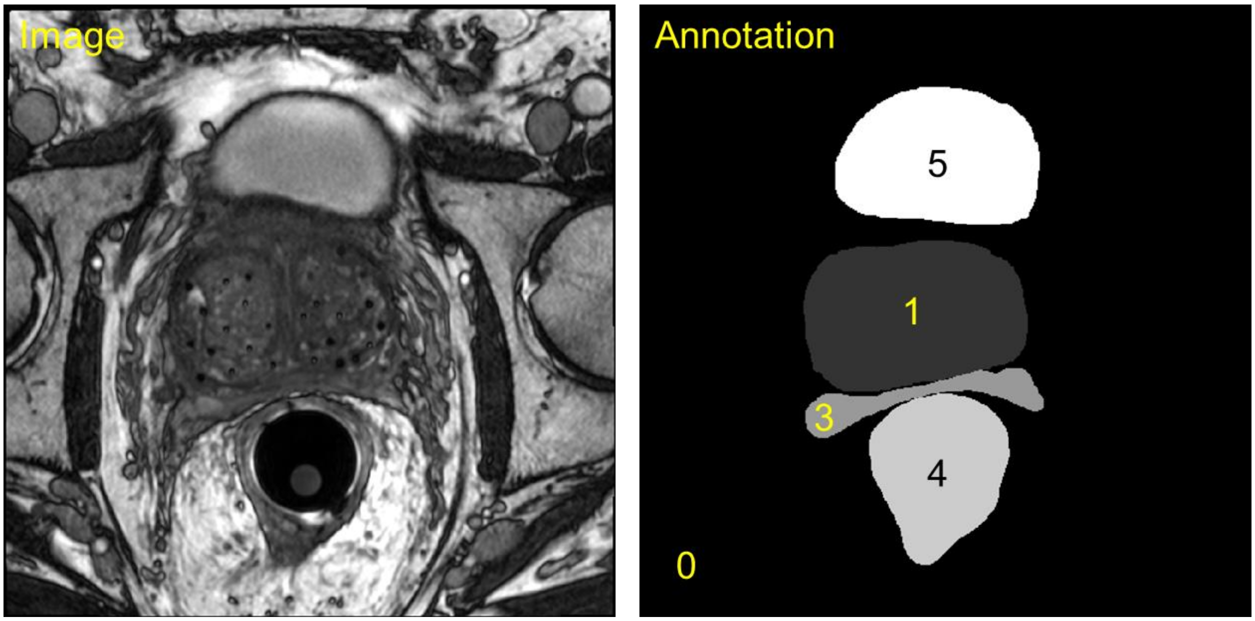

Figure 3. Example of an image/annotation pair for a pixel-wise classification task with FCNs. Numbers correspond to class labels. Class 2 isn't shown in this image but appears in another image in the dataset.

For pixel-wise regression tasks, the annotation for an image will be an array of floating point numbers corresponding to the values the user wishes to regress over. For example, if the user wanted to perform image-to-image regression, then the annotation would be another image of the same size as the input image.

2.3 Generative adversarial networks

The image and annotation pairs for image translation with generative adversarial networks (GANs) are similar to the pairs for FCNs; the input data will be an image and the annotations will an array of pixel-wise labels for that image. For 2D image translation, both the images and annotations should have dimensions (n_{rows}, n_{cols}, $n_{chans1,2}$). For 3D images, both the images and annotations should have dimensions (n_{rows}, n_{cols}, n_{slices}, $n_{chans1,2}$). Note that the number of channels must be equal between the image and annotation for cycleGAN.

2.4 Bounding box detectors

For detection tasks with bounding box detectors (BBDs), the input data will be an image and the annotations will a vector containing the coordinates defining a box around an object and the class of the object. Currently, only 2D images are supported for BBDs. The 2D images should have dimensions (n_{rows}, n_{cols}, n_{chans}). The annotation for an image is an array of size ($_{\text{nbox,i}}$, 5) box coordinates and object classes (4 coordinate variables + 1 class variable = 5) contained within the image, where $n_{box,i}$ is the number of ground truth boxes in image $_i$ ϵ [1, n_{imgs}]. All class numbering schemes should start at 0 and increase linearly with an increment of 1.

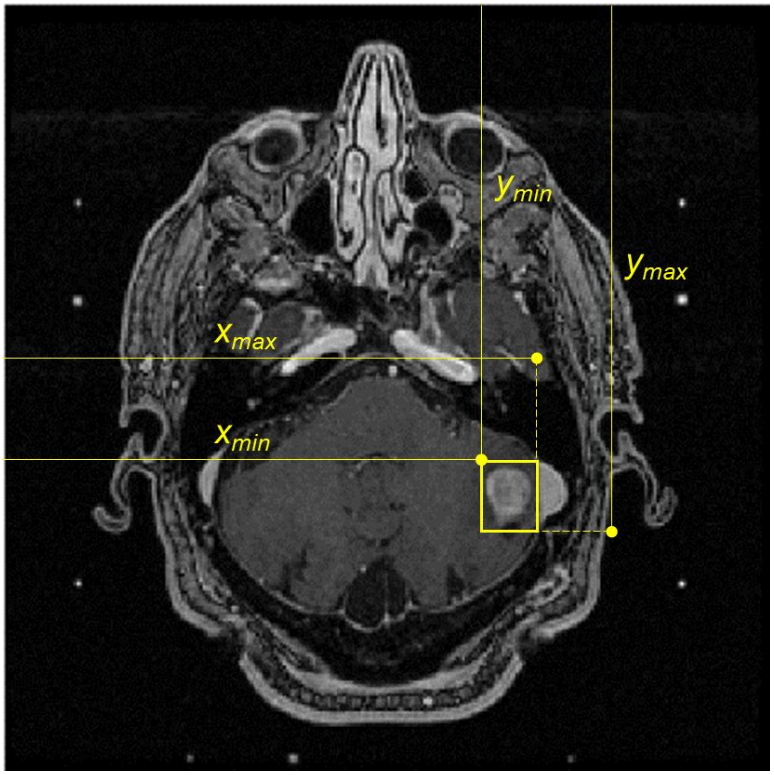

Figure 4. Image with an example of a bounding box coordinate definitions overlaid.

2.5 Structuring and saving data files

The images and annotations should be stored in their own separate HDF5 files. Each image should be stored as a separate dataset within the images HDF5 file. Similarly, each corresponding annotation should be stored as a separate dataset within the annotations HDF5 file. The dataset name for each image/annotation pair should be identical. Furthermore, the dataset names should be integer numbers starting from 0 and increasing linearly to n_{imgs} - 1. An example of saving a matrix of training images and a matrix of corresponding annotations is shown below:

```
import h5py

# Say we have a numpy array 10000 training images. The
# array has size (10000, 512, 512, 1). We want to create
# a DL model to segment these images. So, we also have a
# numpy array containing 10000 segmentation masks. This
# array also has size (10000, 512, 512, 1).

# Open the images HDF5 file to be created.
```

```
f = h5py.File(imgs_save_path, 'w')

# Iterate over the array of images, saving each as a
# separate dataset within the HDF5 file. The dataset
# names are just numbers from 0 to 9999.
for i, img in enumerate(imgs):
    f.create_dataset(str(i), data=img)
f.close()

# Open the annotations HDF5 file to be created.
f = h5py.File(annos_save_path, 'w')

# Iterate over the array of segmentation masks,
# saving each as a separate dataset within the HDF5
# file. The dataset names are just numbers from 0 to
# 9999.
for i, anno in enumerate(annos):
    f.create_dataset(str(i), data=anno)
f.close()
```

Notice that the dataset names in both the images file and the annotations file are exactly the same. Example datasets can be found at https://github.com/jeremiahws/dlae/tree/master/datasets to help users understand the data shapes and file structures for the different types of DL models.

3 Data preprocessing and augmentation

Users have the option to apply a few different preprocessing steps to their data before training DL or making predictions on one. Different type of image normalization are routinely used in the development of DL models. The type of normalization can vary depending on the model being developed. The most common types of normalization are global scaling, squashing values to the range [0, 1], and squashing values to the range [-1, 1].

For global scaling, users can specify the minimum and maximum values of the entire image dataset, which are used to apply a single scaling factor to the entire image dataset. After global normalization, all values in the image dataset will fall inbetween one of two ranges [0, 1] (however, individual images may fall in a smaller range >0 and <1) or [-1, 1] (however, individual images may fall in a smaller range >-1 and <1). The range depends on the type of scaling selected by the user.

It's also possible to apply sample-wise normalization. With this normalization, individual images are normalized using their minimum and maximum values. After normalization, each image will fall into one of two ranges: [0, 1] or [-1, 1]. In either scenario, it is guaranteed that the full range is utilized for each image (unlike global scaling).

Users also have the option to apply augmentation to their data for training DL models. A number of augmentation types can be applied. Examples include affine transformations, brightness adjustments, and ZCA whitening. These augmentation steps only get applied during training if they are used.

4 GUI overview

The GUI is partitioned into many of the tasks encountered in a DL workflow. These tasks include loading, preprocessing, and augmenting datasets, analyzing hyperparameter states, model construction, specification of training configurations, and launching a DL experiment. Each section of the GUI is described in the following sections.

4.1 Home screen

The home screen is displayed the when GUI is launched. It contains access points to all of the sub-menus in the GUI, as well as hosts options for loading in prebuilt models and modifying the layer configurations of models. The GUI home screen is shown in Figure 5.

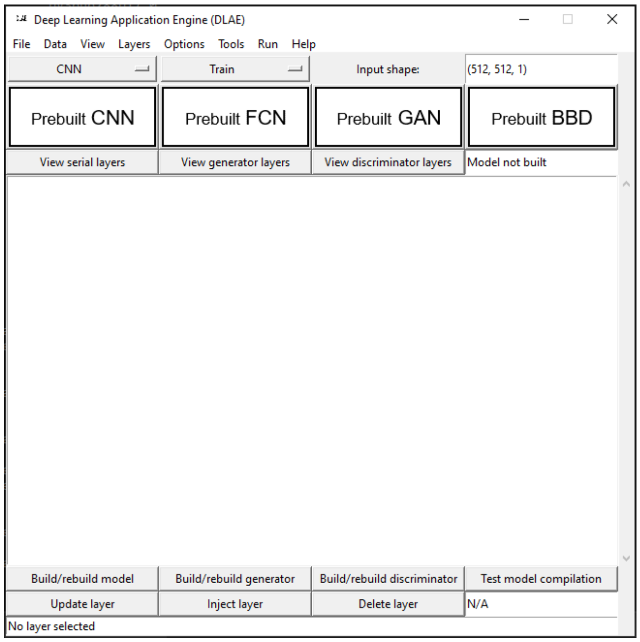

Figure 5. GUI home screen.

Access points to many of the sub-menus are available from the top menu bar. The registered serial (sequentially connected layers) can be viewed by clicking the “View serial layers button”. Similarly, the registered layers for the generator and discriminator of a GAN can be viewed by clicking the associated buttons.

Buttons towards the bottom of the home screen are used for modifying the list of layers that is displayed. There are a few different type of layer modifications that users can perform. First, users can select a layer, modify its parameters, then update the layer in the list. Second, users can select a layer, then create a new layer to be injected after the selected layer. Third, users can select a layer in the box to be deleted. Users should make sure to click the appropriate “Build/rebuild” button after then are finished modifying the list of layers. *Otherwise, the updated list of layers will not be registered.*

Users have the option of loading in a number of prebuilt layer configurations across a range of DL techniques. This is possible by clicking one of the buttons “Prebuilt” buttons near the top of the home screen. As an example, Figure 6 below shows the sub-menu of the prebuilt FCNs that are currently incorporated into DLAE, which was accessed by clicking the “Prebuilt FCN” button.

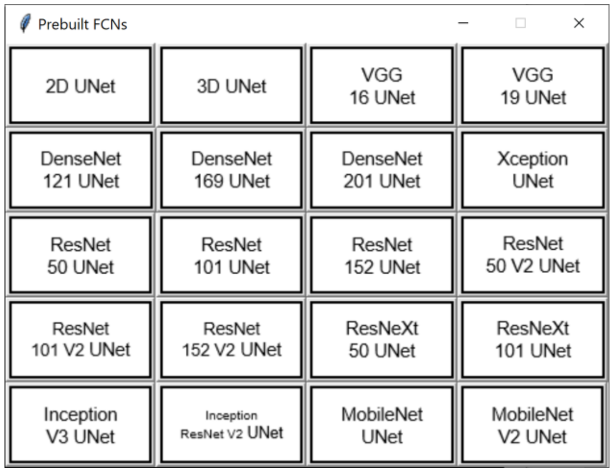

Figure 6. Sub-menu for prebuilt FCNs.

4.2 File menu

Figure 7 (left) shows the available sub-menus for the File menu. The primary purpose of the File menu is to load/save engine configuration files, and load model checkpoints and trained models.

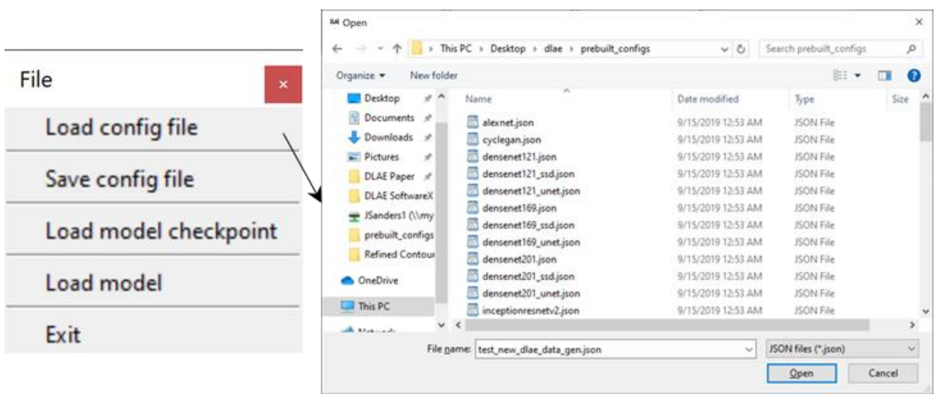

Figure 7. (Left) Sub-menus available from the File menu. (Right) Example of the sub-menu opened after clicking the "Load config file" button.

4.3 Data menu

Figure 8 (left) shows the sub-menus available from the Data menu. Training, validation, and testing datasets can be loaded through dedicated sub-menus. Furthermore, configurations for reprocessing and augmentation of the datasets can be specified through dedicated sub-menus. It is important to distinguish the preprocessing from the augmentation. Preprocessing steps, such as image normalization, are applied to all of the datasets (train, validation, and test). In contrast, augmentation steps, such as affine transformations, are applied only to the training data during a training session to amplify the number of training examples.

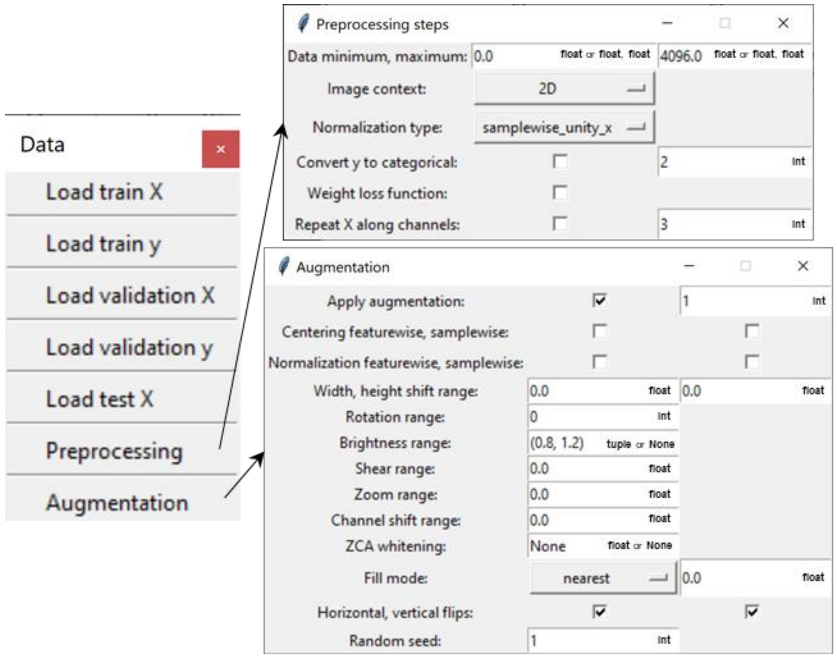

Figure 8. (Left) Sub-menus available from the Data menu. (Right) Sub-menus for data preprocessing and data augmentation. Data types for the parameters are shown in small next towards the right of the entries.

4.4 View menu

Figure 9 shows the GUI parameter states sub-menu. This sub-menu displays all of the parameters in their current states (except for the paths to files). This sub-menu is mainly to allow users to verify they made all of their desired specifications prior to launching the engine.

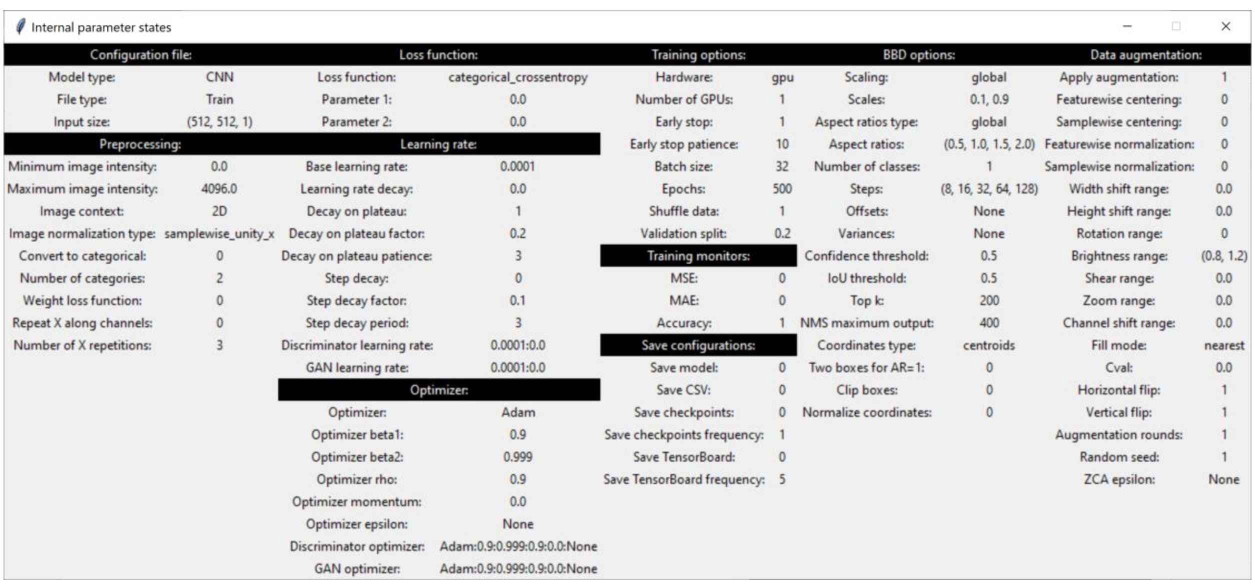

Figure 9. Sub-menu of the GUI parameter states. These parameters are used during execution of the engine.

4.5 Layers menu

Figure 10 (left) shows the sub-menus available from the Layers menu. The functionality provided in these sub-menus are essential for building a custom DL model. Users can use these sub-menus to build up a layer configuration for a custom model, then register the final model from the Home menu. Layers are added one after the other in serial fashion. For GANs, users are able to register separate serial models for both the generator and the discriminator. As mentioned previously, layers that are already added to the layers list in the Home menu can be modified via GUI tools on the Home menu.

Certain models require skip connections or hook connections (for predictor layers in BBDs). Points to inject these layers can be defined in the layers list through buttons in the Utility layers sub-menu.

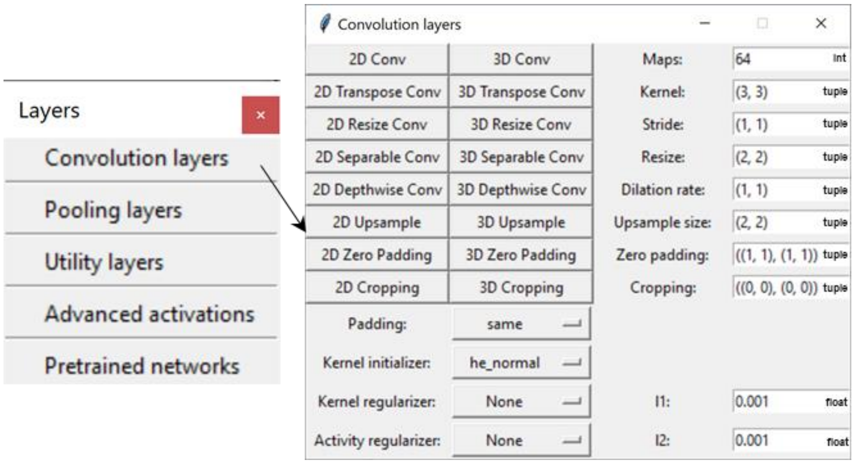

Figure 10. (Left) Sub-menus available from the Layers menu. (Right) Sub-menu for Convolutional layers. Data types for the parameters are shown to the right of the entries.

4.6 Options menu

Figure 11 (left) shows the sub-menus available from the Options menu. These sub-menus allow users to specify a number of options used during a training session, including the loss function to be used. There are two place holders for users to specify parameters to be used for the loss function. Depending on the loss function selected, the engine may use none or both of the parameters selected. Users should view section 10 to see which loss functions require parameter specifications. For example, mean-squared-error requires no parameters, pix2pix requires 1 parameter, and focal loss requires both.

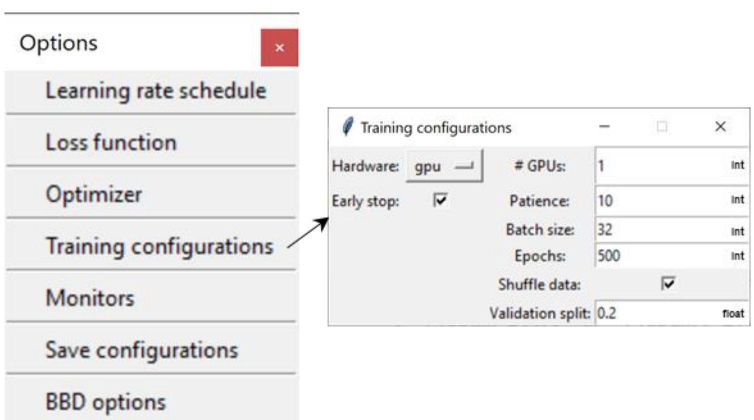

Figure 11. (Left) Sub-menus available from the Options menu. (Right) Training configurations sub-menu. Data types for the parameters are shown to the right of the entries.

4.7 Tools, Run, and Help menus

The Tools menu (Figure 12, left) provides a couple of functionalities. First, users can delete the layer configurations that have been registered by clicking one of the three delete buttons. These were placed in a separate menu from the Home screen to prevent accidental deletion of layer configurations. Second, users can open the most recent TensorBoard log in their browser.

From the Run menu (Figure 12, center) users can launch the engine with the engine configuration structure that has been made in the GUI. Before this, it is recommended that users save the configurations to a configuration file.

The Help menu (Figure 12, right) hosts two primary functions. First, it can direct users to the DLAE Github repository, which is opened in the user's default browser. Second, users can open an error log that displays and errors encountered during execution of the engine.

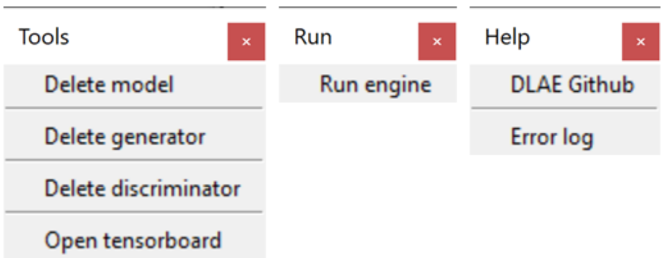

Figure 12. (Left) Tools, (Center) Run, and (Right) Help menus.

5 Configuration files

Configuration files are central to using DLAE. They enable three primary functionalities. First, if users are using DLAE in GUI mode, users can save the model and configurations they have constructed for a specific DL application to a file. They can then reload this file later on or share it with a collaborator. Second, if users are using DLAE in silent mode, they can pass a configuration file to DLAE from the command line to automatically launch a training session. This enables large scale ablation studies and hyperparameter searches through the use of an experiment generator that spawns configuration files defining unique experiments. Experiment generators are covered in a later section. Finally, users can automatically launch inference sessions on a trained model with a configuration file, which enables simplistic integration of trained models into workflows.

The parameters within a configuration file can be modified either by user inputs to the GUI or via programmatic interfacing with configuration files (e.g. an experiment generator). The parameters within a configuration file, along with brief descriptions, are shown below:

<u>config_file:</u>

model_signal (**str**): the type of model the user is working on (CNN, FCN, GAN, or BBD)
type_signal (**str**): the type of configurations that are being developed (training or inference)
input_shape (**tuple**): the input shape of the images

<u>paths:</u>

load_config (**str**): path to the configuration file to be loaded
load_checkpoint (**str**): path to the model checkpoint file to be loaded
load_model (**str**): path to the model file to be loaded
train_X (**str**): path to the training images
train_y (**str**): path to the training annotations

validation_X (**str**): path to the validation images
validation_y (**str**): path to the validation annotations
test_X (**str**): path to the testing images

<u>preprocessing:</u>

minimum_image_intensity (**float** or **float**, **float**): minimum image intensity in the images and (optional) annotations
maximum_image_intensity (**float** or **float**, **float**): maximum image intensity in the images and (optional) annotations
image_context (**str**): type of images being used (2D or 3D)
normalization_type (**str**): type of normalization to be applied to the images and annotations
categorical_switch (**bool**): whether or not to convert the annotations to categoricacl variables
weight_loss_switch (**bool**): whether or not to weight the loss function
repeat_X_switch (**bool**): whether or not to repeat the images along the channels dimension
repeat_X_quantity (**int**): number of repetitions of the image along the channel dimension
categories (**int**): number of categories (or classes)

<u>augmentation:</u>

apply_augmentation_switch (**bool**): whether or not to apply data augmentation
featurewise_centering_switch (**bool**): whether or not to perform featurewise centering
samplewise_centering_switch (**bool**): whether or not to perform samplewise centering
featurewise_normalization_switch (**bool**): whether or not to apply featurewise normalization
samplewise_normalization_switch (**bool**): whether or not to apply samplewise normalization
width_shift (**float**): maximum amount of shifting to apply in the columns dimension in terms of a fraction of the number of columns
height_shift (**float**): maximum amount of shifting to apply in the rows dimension in terms of a fraction of the number of columns
rotation_range (**int**): maximum amount of degree rotation to apply
brightness_range (**tuple** or **None**): range of brightness adjustment to apply
shear_range (**float**): maximum amount of shearing to apply in radians
zoom_range (**float**): maximum amount of zooming in/out of the image
channel_shift_range (**float**): maximum range of random channel shifting to apply
fill_mode (**str**): type of filling to apply to edges of images during augmentation
cval (**float**): value to apply to edges when using “constant” fill mode
horizontal_flip_switch (**bool**): whether or not to randomly flip the image horizontally
vertical_flip_switch (**bool**): whether or not to randomly flip the image vertically
rounds (**int**): amount of amplification of the training data
random_seed (**int**): seed point for the random number generator
zca_epsilon (**float** or **None**): epsilon used in ZCA whitening

<u>loss_function:</u>

loss (**str**): loss function to be used for training
parameter1 (**float**): parameter required for certain loss functions
parameter2 (**float**): parameter required for certain loss functions

<u>learning_rate_schedule:</u>

learning_rate (**float**): learning rate for the optimizer
learning_rate_decay_factor (**float**): decay factor for the learning rate
decay_on_plateau_switch (**bool**): whether or not to decay the learning rate when the validation loss plateaus

decay_on_plateau_factor (**float**): fraction of decay to apply to the learning rate when plateau occurs
decay_on_plateau_patience (**int**): amount of epochs to wait with no improvement in validation loss before decaying the learning rate
step_decay_switch (**bool**): whether or not to apply step decay
step_decay_factor (**float**): fraction of decay to apply
step_decay_period (**int**): how often, in epochs, to apply step decay
discriminator_learning_rate (**float**): learning rate for discriminator
gan_learning_rate (**float**): learning rate for the GAN

<u>optimizer:</u>

optimizer (**str**): type of optimizer to use
beta1 (**float**): parameter used by Adam-type optimizers
beta2 (**float**): parameter used by Adam-type optimizers
rho (**float**): parameter used by Adagrad and RMSProp optimizers
momentum (**float**): parameter used by SGD optimizer
epsilon (**float**): parameter used by some optimizers
discriminator_optimizer (**str**): discriminator optimizer settings, colon delimited
gan_optimizer (**str**): GAN optimizer settings, colon delimited

<u>training_configurations:</u>

hardware (**str**): type of hardware to use (cpu, gpu, or multi-gpu)
number_of_gpus (**int**): number of GPUs for multi-gpu
early_stop_switch (**bool**): whether or not to stop training after no improvement in validation loss
early_stop_patience (**int**): number of epochs to wait before terminating training after no improvement in validation loss
batch_size (**int**): mini-batch size for training and inference
epochs (**int**): number of training epochs before terminating training
shuffle_data_switch (**bool**): whether or not to shuffle the data before training
validation_split (**float**): fraction of training data to reserve for validation

<u>monitors:</u>

mse_switch (**bool**): whether or not to monitor mean-squared-error
mae_switch (**bool**): whether or not to monitor mean-absolute-error
accuracy_switch (**bool**): whether or not to monitor accuracy

<u>save_configurations:</u>

save_model_switch (**bool**): whether or not to save the final model at the end of training
save_model_path (**str**): path to save the model
save_csv_switch (**bool**): whether to save the training history to a CSV file
save_csv_path (**str**): path to save the CSV file of training history
save_checkpoints_switch (**bool**): whether or not to save model checkpoints
save_checkpoints_path (**str**): path to save model checkpoints
save_checkpoints_frequency (**int**): how often, in epochs, to save model checkpoints
save_tensorboard_switch (**bool**): whether or not to save TensorBoard logs
save_tensorboard_path (**str**): path to save TensorBoard logs
save_tensorboard_frequency (**int**): how often, in epochs, to save TensorBoard logs

<u>bbd_options:</u>

scaling_type (**str**): whether to apply scaling globally or per predictor head
scales (**float**, **float** or **tuple**): the scaling for the sizes of the archor boxes
aspect_ratios_type (**str**): whether the aspect ratios are defined globally or per predictor head
aspect_ratios (**tuple**): aspect ratios of the anchor boxes
number_classes (**int**): number of object classes

steps (**tuple** or **None**): separation of archor box centroids
offsets (**tuple** or **None**): offset of upper most anchor box from edge of image
variances (**tuple**): values to divide anchor box offsets by
confidence_threshold (**float**): confidence threshold for non-max suppression
iou_threshold (**float**): IoU threshold for non-max suppression
top_k (**int**): top k number of objects to be kept after non-max suppression
nms_maximum_output (**int**): maximum number of boxes to keep from non-max suppression
coordinates_type (**str**): the formatting of the input box coordinates
two_boxes_for_AR1_switch (**bool**): whether or not to predict two boxes for aspect ratios equal to one
clip_boxes_switch (**bool**): whether or not to clip boxes outside the image
normalize_coordinates_switch (**bool**): whether or not to normalize the image coordinates
positive_iou_threshold (**float**): IoU threshold for positive examples
negative_iou_limit (**float**): IoU threshold for negative (background) examples

<u>layers:</u>

serial_layer_list (**list**): layers for a serial model (every model but a GAN)
generator_layer_list (**list**): generator layers for a GAN
discriminator_layer_list (**list**): discriminator layers for a GAN

6 Layer definitions

Layers of a model are added in series and display in the home screen (Figure 13). 2D and 3D versions of the same layer type are similar, they just require an additional parameter where appropriate (e.g. kernel of (3, 3) for 2D would be (3, 3, 3) for 3D). The specifications for each layer are defined below in the following format:

<u>**Layer type:**</u>
Layer
Parameter definitions
Parameter data types

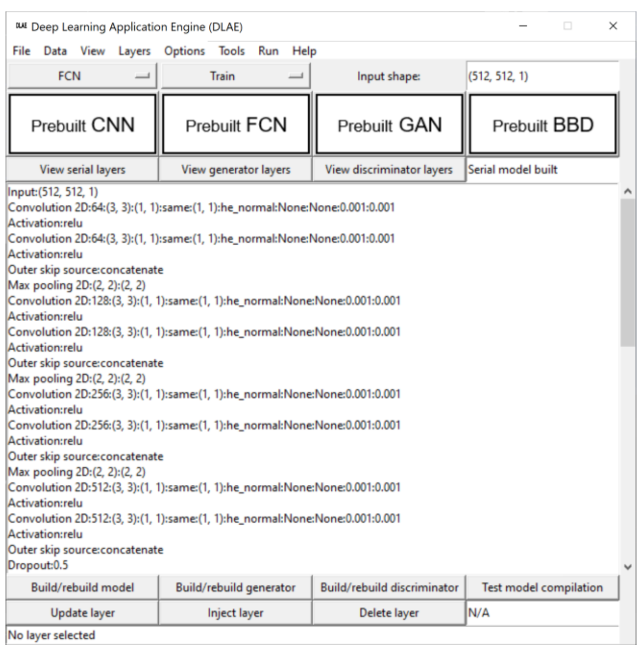

Figure 13. Home screen with a populated list of layers.

Input layer:
Input:(512, 512, 1)
Layer name:shape
str:tuple

Pretrained network layer:
InceptionResNetV2:False:none:(512, 512, 1):True:False
Network type:keep classifier heads:weights:input shape:skip connection starts:hook connections
str:bool:str:tuple:bool:bool

Zero padding layer:
Zero padding 2D:((1, 1), (1, 1))
Layer name:padding
str:tuple

Max pooling layer:
Max pooling 2D:(2, 2):(2, 2)
Layer name:pooling:stride
str:tuple:tuple

Convolutional layer:
Convolution 2D:1024:(3, 3):(1, 1):same:(1, 1):he_normal:None:None:0.001:0.001
Layer name:feature maps:kernel:stride:padding:dilation:initializer:kernel reg:activity reg:l1 val:l2 val
str:int:tuple:tuple:str:tuple:str:str or None:str or None:float:float

Batch normalization layer:
Batch normalization:0.99:0.001
Layer name:momentum:epsilon
str:float:float

Activation layer:
Activation:relu
Layer name:activation type
str:str

Resize convolutional layer:
Resize convolution 2D:512:(3, 3):(1, 1):(2, 2):same:(1, 1):he_normal:None:None:0.001:0.001
Layer name:feature maps:kernel:stride:resize:padding:dilation:initializer:kernel reg:activity reg:l1 val:l2 val
str:int:tuple:tuple:tuple:str:tuple:str:str or None:str or None:float:float

Skip connection layer:
Outer skip source:concatenate and Outer skip target:concatenate
Layer name:skip type
str:str

Advanced activation layer:
Leaky reLU:0.2
Layer name:activation parameter
str:float

Transpose convolutional layers:
Transpose convolution 2D:64:(3, 3):(1, 1):same:(1, 1):he_normal:None:None:0.001:0.001
Layer name:feature maps:kernel:stride:padding:dilation:initializer:kernel reg:activity reg:l1 val:l2 val
str:int:tuple:tuple:str:tuple:str:str or None:str or None:float:float

Separable convolutional layer:
Separable convolution 2D:64:(3, 3):(1, 1):same:(1, 1):he_normal:None:None:0.001:0.001
Layer name:feature maps:kernel:stride:padding:dilation:initializer:kernel reg:activity reg:l1 val:l2 val
str:int:tuple:tuple:str:tuple:str:str or None:str or None:float:float

Depthwise separable convolutional layer:
Depthwise separable convolution 2D:64:(3, 3):(1, 1):same:he_normal:None:None:0.001:0.001
Layer name:feature maps:kernel:stride:padding:dilation:initializer:kernel reg:activity reg:l1 val:l2 val
str:int:tuple:tuple:str:tuple:str:str or None:str or None:float:float

Upsampling layer:
Upsample 2D:(2, 2)
Layer name:upsampling
str:tuple

Cropping layer:
Cropping 2D:((0, 0), (0, 0))
Layer name:cropping
str:tuple

Average pooling layer:
Average pooling 2D:(2, 2):(2, 2)
Layer name:pool size:stride
str:tuple:tuple

Global max pooling layer:
Global max pooling 2D
Layer name
str

Global average pooling layer:
Global average pooling 2D
Layer name
str

Reshape layer:
Reshape:(262144, 1)
Layer name:resize
str:tuple

Dropout layer:
Dropout:0.5
Layer name:dropout rate
str:float

Dense layer:
Dense:1024
Layer name:neurons
str:int

Flatten layer:
Flatten
Layer name

str

Permute layer:
Permute:(1, 0, 2)
Layer name:permutation
str:tuple

Spatial dropout layer:
Spatial dropout 2D:0.5
Layer name:dropout rate
str:float

Gaussian dropout layer:
Gaussian dropout:0.5
Layer name:dropout rate
str:float

Alpha dropout layer:
Alpha dropout:0.5
Layer name:dropout rate
str:float

Gaussian noise layer:
Gaussian noise:0.1
Layer name:noise standard deviation
str:float

Hook connection layer:
Hook connection source and Hook connection target
Layer name
str

Inner skip connection layer:
Inner skip source:concatenate and Inner skip target:concatenate
Layer name:skip type
str:str

7 Engine overview

This section gives a brief overview of the engine. The engine executes after provided a set of engine configurations. The engine configurations can either be created from the GUI interface or loaded from a configuration file. In either case, the engine configurations will be used to create an engine configuration structure. A high level overview of the engine configuration structure is shown below:

```
class EngineConfigurations(object):
    def __init__(self, configs):
        self.dispatcher = Dispatcher(configs)
        self.data_preprocessing = Preprocessing(configs,
                                                self.dispatcher)
        self.augmentation = Augmentation(configs)
        self.train_options = TrainingOptions(configs)
        self.train_data = TrainData(configs,
                                    self.data_preprocessing,
```

```
                                self.dispatcher,
                                self.augmentation,
                                self.train_options)
        self.val_data = ValidationData(configs,
                                   self.data_preprocessing,
                                   self.dispatcher,
                                   self.augmentation,
                                   self.train_options,
                                   self.train_data)
        self.test_data = TestData(configs,
                              self.data_preprocessing,
                              self.dispatcher,
                              self.augmentation,
                              self.train_options)
        self.learning_rate = LearningRate(configs)
        self.optimizer = Optimizer(configs, self.learning_rate)
        self.monitors = Monitors(configs)
        self.loader = Loader(configs)
        self.saver = Saver(configs)
        self.layers = Layers(configs)
        self.loss_function = LossFunction(configs,
                                          self.data_preprocessing)
        self.callbacks = Callbacks(self.saver,
                                   self.learning_rate,
                                   self.train_options)
```

The dispatcher class extracts the type of DL technique to apply (CNN, FCN, GAN, or BBD) and whether the configurations are for a training session or an inference session. The preprocessing class controls the preprocessing steps to apply to the data, such as intensity normalization. The augmentation class controls the type of data augmentation to apply to the training data. The training options class contains the training parameters to use during a training session (epochs, batch size, etc.). The training, validation, and testing data classes take the preprocessing, augmentation, and training options classes and use them to construct dataset generators that are used during training and inference sessions. An example dataset generator for an FCN is shown below:

```
generator = FCN2DDatasetGenerator(
    self.s_trainXPath,
    self.s_trainYPath,
    rotation_range=augmentation.i_rotation_range,
    width_shift_range=augmentation.f_width_shift,
    height_shift_range=augmentation.f_height_shift,
    shear_range=augmentation.f_shear_range,
    zoom_range=augmentation.f_zoom_range,
    flip_horizontal=augmentation.b_horizontal_flip,
    flip_vertical=augmentation.b_vertical_flip,
    featurewise_center=augmentation.b_fw_centering,
    featurewise_std_normalization=augmentation.b_fw_normalization,
    samplewise_center=augmentation.b_sw_centering,
    samplewise_std_normalization=augmentation.b_sw_normalization,
    zca_epsilon=augmentation.f_zca_epsilon,
    shuffle_data=train_optiions.b_shuffleData,
    rounds=augmentation.i_rounds,
    fill_mode=augmentation.s_fill_mode,
    cval=augmentation.f_cval,
```

```
        interpolation_order=1,
        seed=augmentation.i_random_seed,
        batch_size=train_optiions.i_batchSize,
        validation_split=train_optiions.f_validationSplit,
        subset='train',
        normalization=preprocessing.s_normalization_type,
        min_intensity=minimums,
        max_intensity=maximums,
        categorical_labels=preprocessing.b_to_categorical,
        num_classes=preprocessing.i_num_categories,
        repeat_chans=preprocessing.b_repeatX,
        chan_repititions=preprocessing.i_repeatX)
```

The dataset generator will load images on the fly during training or inference, rather than loading all of the images into memory at once. This enables users to developed models using large datasets where all of the images cannot fit into memory at once. Along with the dataset classes, the learning rate, optimizer, monitors, layers, loss function, and callbacks classes are used in executing the graph computations. All of the model types have a dedicated class with the same set of methods. An example class for an FCN is shown below:

```
class FullyConvolutionalNetwork(object):
    def __init__(self, engine_configs):
        self.engine_configs = engine_configs
        self.graph = tf.get_default_graph()
        self.model = None
        self.parallel_model = None

    def construct_graph(self):
        # graph construction
        pass

    def compile_graph(self):
        # graph compilation
        pass

    def train_graph(self):
        # train graph
        pass

    def predict_on_graph(self):
        # make predictions on graph
        pass
```

The construct graph method builds the model with the set of layers that are defined. The compile graph method compiles the model with the specified optimizer, learning rate schedule, and loss function. If a training session is being performed, the train graph method will be executed. Similarly, if an inference session is being performed, the predict on graph method will be called.

8 Building custom models

Custom models can be constructed through GUI controls or with an experiment generator (a later section). Users without much programming experience may choose to use the GUI. All models must start with an input layer, from the Utility layers sub-menu, that defines the shape of the input image. From that, a layer configuration can be built on top of the input layer for the specific type of DL technique the user wishes to apply. The type of DL technique to be applied (CNN, FCN, GAN, or BBD) must be specified in the drop down menu on the home screen. Users can then specify the training options, loss function, learning rate, and optimizer to use during training. Data augmentation and preprocessing steps to apply can be selected in the Data menu. A few example custom applications are presented in a later section.

9 Utilizing prebuilt models and transfer learning

Prebuilt models can be easily loaded from one of the Home screen buttons. They can also be modified based on the user's preferences. Prebuilt networks (those developed on the ImageNet dataset) can be used as convolutional feature extractors across all the DL techniques in DLAE. For CNNs, dense layers can be applied on top of the convolutional feature extractor to make class predictions or to regress over a set of coordinates. For FCNs, the prebuilt networks can be used as the convolutional encoding function, from which users can build a convolutional decoding function on top of. In a similar fashion, users can use the prebuilt networks to construct the generator function and/or the discriminator function. Finally, the prebuilt netwroks can be used as convolutional feature extractors for FCN-based BBDs. Users can either use weights pretrained on ImageNet or start with randomly initialized kernels.

10 Objective functions

Currently, there are 10 objective functions included in DLAE. Brief descriptions of each function are shown below. These functions can be supplemented in the future.

10.1 Crossentropy

For image classification with CNNs and pixel-wise classification with FCNs, users can choose to use crossentropy, defined as

$$\mathcal{L}_{CE}(y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i),$$

where y_i and $\hat{y}_i$ are the ground truth and predicted classes, respectively. When using CE as the objective function, DLAE will minimize the CE between the ground truth and predicted classes. For problems where there are multiple classes (such as multi-organ pixel-wise segmentation), the class labels can be converted to categorical variables within DLAE, and DLAE will minimize the CE across classes during training.

10.2 Mean squared error

For image regression with CNNs and pixel-wise regression with FCNs, users can choose to use mean squared error, defined as

$$\mathcal{L}_{MSE}(y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

When using MSE as the objective function, DLAE will minimize the MSE between the ground truth and predicted variables.

10.3 Mean absolute error

For image regression with CNNs and pixel-wise regression with FCNs, users can choose to use mean absolute error, defined as

$$\mathcal{L}_{MAE}(y) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|.$$

When using MAE as the objective function, DLAE will minimize the MAE between the ground truth and predicted variables.

10.4 Tversky index

For image segmentation with FCNs, users can choose to use Tversky index [1], defined as

$$\mathcal{L}_T(A, B; \alpha, \beta) = \frac{|AB|}{|AB| + \alpha|A/B| + \beta|B/A|},$$

where A is the ground truth segmentation mask, B is the predicted segmentation mask, and $\alpha \in [0, 1]$ and $\beta \in [0, 1]$ are parameters which control the penalties on the false positives (FPs) and false negatives (FNs). When $\alpha = \beta = 0.5$, the Tversky index is equivalent to the Dice coefficient (or F_1 score). When using Tversky index as the objective function, DLAE will minimize the Tversky index between the ground truth and predicted segmentation masks.

10.5 Condictional GAN + L1

For paired image-to-image translation with conditional adversarial networks, users can chose to use conditional GAN + L1 loss [2], defined as

$$\mathcal{L}_{pix2pix}(G, D) = \mathcal{L}_{cGAN}(G, D) + \lambda\mathcal{L}_{L1}(G),$$

where

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}\,[\log D(x, y)] + \mathbb{E}_{x,z}\,\left[\log\left(1 - D\big(x, G(x, z)\big)\right)\right],$$

is the objective function for a conditional GAN and

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}\,[\|y - G(x, z)\|_1],$$

is an additional L1 penalty enforced on the pixels of the images generated by G, which attempts to perform the mapping G: $X \to Y$, where X and Y have paired pixel-wise annotations. The additional L1 penalty produces less blurring than with an L2 penalty on the pixels. In this scenario, both G and D are neural networks competing in a minimax game, and DLAE will play the minimax game $\min_G \max_D \mathcal{L}_{pix2pix}(G, D)$.

10.6 Adversarial + cycle consistency

For unpaired image-to-image translation with cycle-consistent adversarial networks, users can choose to use adversarial + cycle consistency loss [3], defined as:

$$\mathcal{L}_{cycleGAN}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda\mathcal{L}_{cyc}(G, F),$$

where

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_y\,[\log D_Y(y)] + \mathbb{E}_x\,\left[\log\left(1 - D_Y\big(G(x)\big)\right)\right],$$

is the objective function for the mapping G: $X \to Y$ with discriminator D_Y,

$$\mathcal{L}_{GAN}(F, D_X, Y, X) = \mathbb{E}_x\,[\log D_X(x)] + \mathbb{E}_y\,\left[\log\left(1 - D_X\big(F(y)\big)\right)\right],$$

is the objective function for the mapping F: $Y \to X$ with discriminator D_X, and

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_x\,\left[\left\|F\big(G(x)\big) - x\right\|_1\right] + \mathbb{E}_y\,\left[\left\|G\big(F(y)\big) - y\right\|_1\right].$$

is a cycle consistency loss enforcing forward ($x \to G(x) \to F(G(x)) \approx x$) and backward ($y \to F(y) \to G(F(y)) \approx y$) cycle consistencies. In this scenario, G, F, D_Y, and D_X are all neural networks competing in a minimax game, and DLAE will play the minimax game $\min_{G,F} \max_{D_X,D_Y} \mathcal{L}_{cycleGAN}(G, F, D_X, D_Y)$.

10.7 SSD loss

For object detection with SSDs, users can choose to use SSD loss [4], defined as

$$\mathcal{L}_{SSD}(x,c,l,g) = \frac{1}{N}\left(\mathcal{L}_{conf}(x,c) + \alpha\mathcal{L}_{loc}(x,l,g)\right),$$

where *N* is the number of matched default boxes and α is a weighting term. The SSD loss is a weighted sum of the confidence loss $\mathcal{L}_{conf}$ and localization loss $\mathcal{L}_{loc}$. The confidence loss is a softmax classification loss over the object classes in the training dataset

$$\mathcal{L}_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{ij}^{p}\log\left(\hat{c}_i^{p}\right) - \sum_{i\in Neg}\log\left(\hat{c}_i^{0}\right),$$

where

$$\hat{c}_i^{p} = \frac{\exp\left(c_i^{p}\right)}{\sum_p \exp\left(c_i^{p}\right)}.$$

The localization loss is a Smooth L1 loss between the predicted bounding box *l* and the ground truth bounding box *g*

$$\mathcal{L}_{loc}(x,l,g) = \sum_{i\in Pos}^{N}\sum_{m\in\{cx,cy,w,h\}} x_{ij}^{k}\mathrm{smooth}_{L1}\left(l_i^{m} - \hat{g}_j^{m}\right),$$

where

$$\hat{g}_j^{cx} = \frac{g_j^{cx} - d_i^{cx}}{d_i^{w}},$$

is the contribution to the localization loss from the center offset *cx* to the default bounding box *d* in the width *w* direction,

$$\hat{g}_j^{cy} = \frac{g_j^{cy} - d_i^{cy}}{d_i^{h}},$$

is the contribution to the localization loss from the center offset *cy* to the default bounding box *d* in the height *h* direction,

$$\hat{g}_j^{w} = \log\left(\frac{g_j^{w}}{d_i^{w}}\right),$$

is the contribution to the localization loss from the magnitude of the width *w* of the predicted box, and

$$\hat{g}_j^{h} = \log\left(\frac{g_j^{h}}{d_i^{h}}\right),$$

is the contribution to the localization loss from the magnitude of the height *h* of the predicted box. When using SSD loss as the objective function, DLAE will minimize the SSD loss between the ground truth and predicted bounding boxes and class labels.

10.8 Jaccard distance

For image classification with CNNs and pixel-wise classification with FCNs, users can choose to use Jaccard distance, defined as

$$\mathcal{L}_J(A,B) = 1 - \frac{|A\cap B|}{|A| + |B| - |A\cap B|},$$

where A is the ground truth segmentation mask, B is the predicted segmentation mask. When using Jaccard distance as the objective function, DLAE will minimize the Jaccard distance between the ground truth and predicted annotations.

10.9 Focal loss

For image classification with CNNs and pixel-wise classification with FCNs, users can choose to use focal loss [5], defined as

$$\mathcal{L}_{FL}(p_t) = -\alpha_t(1-p_t)^{\gamma}\log(p_t),$$

where α_t is a balancing factor and γ is a focusing parameter. When using focal loss as the loss function, DLAE will minimize the focal loss between the ground truth and predicted annotations.

10.10 Soft dice

For image classification with CNNs and pixel-wise classification with FCNs, users can choose to use Jaccard distance, defined as

$$\mathcal{L}_D(A, B; \epsilon) = 1 - \frac{|A \cap B| + \epsilon}{|A| + |B| + \epsilon},$$

where A is the ground truth segmentation mask, B is the predicted segmentation mask, and ϵ is a smoothing factor. When using soft Dice as the objective function, DLAE will minimize the soft Dice distance between the ground truth and predicted annotations.

11 Experiment generation

Experiment generators can be used to process several DL experiments on a data set. This allows users to investigate how different models constructs, layer configurations, and hyperparameters affect model performance. Creating an experiment generator is a straightforward process. Rather than changing engine configuration parameters via the GUI, users can load in a configuration file then programmatically modify the engine configurations. The general approach to constructing an experiment generator is to create a predefined set of parameters to run experiments on, then iterate over all the parameters creating an engine configuration structure for each unique set of parameters. A high-level overview an example experiment generator is shown below. This example loads in a baseline segmentation model then iterates over the convolutional encoder and loss function.

```
def main(FLAGS):
    # define the experiments
    encoders = FLAGS.encoders.split(',')
    losses = FLAGS.losses.split(',')
    alpha = FLAGS.loss_param1.split(',')
    experiments = [encoders, losses, alpha]

    # switch to activate training session
    do_train = True

    for experiment in itertools.product(*experiments):
        # load the base configurations
        configs = load_config(
                os.path.join(FLAGS.base_configs_dir,
                             'vgg16_unet.json'))

        # apply some augmentation
        configs['augmentation']['apply_augmentation_switch'] = 'False'
        configs['augmentation']['width_shift'] = '0.15'
        configs['augmentation']['height_shift'] = '0.15'
        # ...

        # set the training configurations
        configs['training_configurations']['batch_size'] = \
            '{}'.format(FLAGS.batch_size)
        # ...
```

```
# modify the encoding function
layers = configs['layers']['serial_layer_list']
input = layers[0].split(':')
input_parts[-1] = '({}, {}, {})'.format(FLAGS.height,
                                        FLAGS.width,
                                        FLAGS.channels)
input = ':'.join(input_parts)
configs['config_file']['input_shape'] = \
    '({}, {}, {})'.format(FLAGS.height,
                          FLAGS.width,
                          FLAGS.channels)
encoder = layers[1]
decoder = layers[2:]
last_conv = decoder[-3]
last_conv_parts = last_conv.split(':')
last_conv_parts[1] = '{}'.format(FLAGS.classes)
last_conv = ':'.join(last_conv_parts)
decoder[-3] = last_conv
encoder_parts = encoder.split(':')
encoder_parts[0] = experiment[0]
encoder_parts[3] = '({}, {}, {})'.format(FLAGS.height,
                                         FLAGS.width,
                                         FLAGS.channels)
if FLAGS.use_skip_connections:
    encoder_parts[-2] = 'True'
else:
    encoder_parts[-2] = 'False'
    while 'Outer skip target:concatenate' in decoder:
        decoder.remove('Outer skip target:concatenate')
encoder = ':'.join(encoder_parts)

layers = []
layers.extend([input, encoder])
for layer in decoder: layers.append(layer)
configs['layers']['serial_layer_list'] = layers

# ensure xentropy only used once per encoder
if experiment[1] == 'categorical_crossentropy':
    if experiment[2] == '0.3':
        configs['loss_function']['loss'] = experiment[1]
    else:
        do_train = False
elif experiment[1] == 'tversky':
    configs['loss_function']['loss'] = experiment[1]
    configs['loss_function']['parameter1'] = experiment[2]
    configs['loss_function']['parameter2'] = \
        str(1. - literal_eval(experiment[2]))
else:
    do_train = False

if do_train is True:
    engine = Dlae(configs)
    engine.run()
```

The full example experiment generator for segmenting images can be found at https://github.com/jeremiahws/dlae/blob/master/fcn_experiment_generator.py.

12 Example applications

Image classification and regression with CNNs

Post-implant prostate MR images were collected. A cuboid bounding box surrounding the prostate was extracted from each of the images. A sliding-window algorithm was used on the ROI to extract 3D sub-windows of size 13 × 13 × 7. These sub-windows, which contained background sub-windows and seed sub-windows, were used to train two CNNs. The annotations for the first CNN were binary labels, 0 for background and 1 for seed. The annotations for the second CNN were centroid locations of the seeds within the individual sub-windows. CNNs with the following layer configurations were constructed:

```
["Input:(13, 13, 7, 1)",
 "Convolution 3D:64:(3, 3, 3):(1, 1, 1):same:(1, 1,
      1):he_normal:None:None:0.001:0.001",
 "Batch normalization:0.99:0.001",
 "Activation:relu",
 "Max pooling 3D:(2, 2, 2):(2, 2, 2)",
 "Convolution 3D:128:(3, 3, 3):(1, 1, 1):same:(1, 1,
      1):he_normal:None:None:0.001:0.001",
 "Batch normalization:0.99:0.001",
 "Activation:relu",
 "Max pooling 3D:(2, 2, 2):(2, 2, 2)",
 "Convolution 3D:256:(3, 3, 3):(1, 1, 1):same:(1, 1,
      1):he_normal:None:None:0.001:0.001",
 "Batch normalization:0.99:0.001",
 "Activation:relu",
 "Flatten",
 "Dense:2048",
 "Batch normalization:0.99:0.001",
 "Activation:relu",
 "Dropout:0.5",
 "Dense:2",
 "Activation:softmax"]
```

The layer configuration above is for the classifier. For the localizer, the number of outputs was changed to 3 and the softmax was removed. Crossentropy loss was used to train the classifier, and mean squared error was used to train the localizer. Configuration files for both models with all other parameters specified can be found at https://github.com/jeremiahws/dlae/tree/master/configs.

Image segmentation with FCNs

The same MR images from the above application were used. A physician manually contoured the prostate, rectum, bladder, and rectum. These contours were used to create segmentation masks of the images. A convolutional encoder developed on the ImageNet data set (Xception) with randomly initialized kernels was used as the convolutional feature extractor. The encoding was upsampled back to the original image resolution by constructing a decoder with a series of resize convolutions. The inputs to the model were the images and the outputs were the segmentation masks. The model was trained by minimizing pixel-wise crossentropy between the predicted and ground truth segmentation masks. A configuration file with all parameters specified can be found at https://github.com/jeremiahws/dlae/tree/master/configs.

Image synthesis with GANs

DCE-MRI and DSC-MRI of the brain were acquired at 60 time points. An anatomical T1 map was acquired in-between the two perfusion scnas, directly after the DCE-MRI. The DSC-MRI was registered to the anatomical T1 map using SPM. NordicIce was used to compute relative cerebral blood volume (rCBV) maps from the DSC-MRI. A conditional GAN was constructed to synthesize the rCBV maps from the DSC-MRI. A UNet FCN was constructed as the generator. A classification CNN was constructed as the discriminator. The first 30 time points of the DCE-MRI were used as 30 channel inputs to the generator. A conditional GAN + L1 loss was used for training. A configuration file with all parameters specified can be found at https://github.com/jeremiahws/dlae/tree/master/configs.

Object detection with BBDs
Post-contrast T1 weighted MR images were acquired for patients undergoing treatment planning for stereotactic radiosurgery. A neuroradiologist manually identified brain metastases in the images. A treating radiation oncologist segmented the metastases to be planned for radiation treatment. Segmentation masks of the brain metastases were used to construct bounding boxes around each metastasis via connected components analysis. A BBD with predictor heads at multiple resolution scales was constructed. An SSD loss was used to train the BBD to predict bounding boxes around the metastases with an associated confidence. A configuration file with all parameters specified can be found at https://github.com/jeremiahws/dlae/tree/master/configs.

Generic examples
We have also created generic examples of developing CNNs, FCNs, GANs, and BBDs for the example data provided. Configuration files for these examples can be found at under the names “example_cnn.json”, “example_fcn.json”, “example_gan.json” , and “example_bbd.json” at https://github.com/jeremiahws/dlae/tree/master/configs. Users will only need to change the paths to where the data is stored locally on their computer.

13 References

[1] Salehi, S.S.M., Erdogmus, D., Gholipour, A., 2017. Tversky loss function for image segmentation using 3D fully convolutional deep networks. arXiv:1706.05721. https://arxiv.org/abs/1706.05721.
[2] Isola, P., Zhu, J.Y., Zhou, T., Efros, A.A., 2016. Image-to-image translation with conditional adversarial networks. arXiv:1611.07004. https://arxiv.org/abs/1611.07004.
[3] Zhu, J.Y., Park, T., Isola, P., Efros, A.A., 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. arXiv:1703.10593. https://arxiv.org/abs/1703.10593.
[4] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., et al., 2015. SSD: Single shot MultiBox detector. arXiv:1512.02325. https://arxiv.org/abs/1512.02325.
[5] Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollár, P., 2017. Focal loss for dense object detection. arXiv:1708.02002. https://arxiv.org/abs/1708.02002.