NVIDIA CLARA TRAIN SDK: TRANSFER LEARNING

DU-09302-002 _v2.0 | March 2019

Getting Started Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

NVIDIA's Clara Train SDK: Transfer Learning toolkit is a python-based SDK that allows developers looking into faster implementation of industry specific Deep Learning solutions to leverage optimized, ready-to-use, pretrained models built in-house by NVIDIA. These pre-trained models accelerate the developer's deep learning training process and reduce higher costs associated with large scale data collection, labeling, and training models from scratch.

This toolkit offers an end-to-end deep learning workflow for accelerating Deep Learning training and inference for Medical Imaging use cases. The models provided are fully trained for Medical Imaging specific reference use cases such as organ and tumor segmentation and classification.

Following pre-trained models are available to download for specific classification and segmentation use cases. Complete details and accuracy metrics are available in the Appendix section of this guide.

- Brain Tumor segmentation
- Liver and Tumor segmentation
- Hippocampus segmentation
- Lung Tumor segmentation
- Prostate segmentation
- Left Atrium segmentation
- Pancreas and Tumor segmentation
- Colon Tumor segmentation
- Hepatic Vessel segmentation
- Spleen segmentation
- Chest X-ray classification

Chapter 2. INSTALLATION

Using the Transfer Learning Toolkit for Medical Imaging requires the following:

Hardware Requirements

Recommended

- 1 GPU or more
- 16 GB GPU memory
- 8 core CPU
- 32 GB system RAM
- 80 GB free disk space

Software Requirements

- Ubuntu 16.04 LTS
- NVIDIA GPU driver v410.xx or above
- nvidia-docker 2.0 installed, instructions: https://github.com/NVIDIA/nvidia-docker.
- Cuda runtime/Python package/Tensorflow These are required, but there is no need to install them. They are included in the docker image.

Early Access registration

Get an NGC API Key

- NVIDIA GPU Cloud account and API key - https://ngc.nvidia.com/
    1. Go to NGC and search for Clara Train container in the **Catalog** tab. This message is displayed, **Sign in to access the PULL feature of this repository**.
    2. Enter your email address and click **Next** or click **Create an Account**.
    3. Choose **nvtltea/med** when prompted for your Organization/Team in the **Set Your Organization** dialog.
    4. Click **Sign In**.

5. Click the **Clara Train SDK** tile.

Save the API key in a secure location. You will need it to use the AI assisted annotation SDK.

Download the docker container

- Execute **`docker login nvcr.io`** from the command line and enter your username and password.
  - Username: $oauthtoken
  - Password: API_KEY
- Execute **`docker pull nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3`**

2.1. Running the container

Once downloaded, run the docker using this command:

```
docker run --runtime=nvidia -it --rm nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3 /
bin/bash
```

If you are on a network that uses a proxy server to connect to the Internet, you can provide proxy server details when launching the container.

```
 docker run --runtime=nvidia -it --rm -e
 HTTPS_PROXY=https_proxy_server_ip:https_proxy_server_port -e
 HTTP_PROXY=http_proxy_server_ip:http_proxy_server_port nvcr.io/nvtltea/
med/tlt-annotate:v0.1-py3 /bin/bash
```

The docker, by default, starts in the **`/workspace`** folder. To access local directories from within the docker, they have to be mounted in the docker. To mount a directory, use the **`-v <source_dir>:<mount_dir>`** option. For more, see Bind Mounts. Here is an example:

```
docker run --runtime=nvidia -it --rm -v /home/<username>/tlt-experiments:/
workspace/tlt-experiments nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3 /bin/bash
```

This mounts the **`/home/<username>/tlt-experiments`** directory in your disk to **`/workspace/tlt-experiments`** in docker.

2.2. Downloading the models

Use the **`tlt-pull`** command to download the models from the NGC model registry:

```
tlt-pull -m $MODEL_NAME -v $VERSION -k $API_KEY -d ./path/to/save/model -o
 nvtltea -t med
```

The **`--version`** argument is a mandatory. Please use **`--list_versions`** to find the all the versions that are available.

Example output from using this command.

```
tlt-pull -k $API_KEY -m segmentation_hippocampus -v 1 -d /var/tmp/test_models/
hippocampus -o nvtltea -t med

3 files to download, total size - 654.38 MB
Downloaded 0 B, 0 files in 0s Download speed: 0 B/s
Downloaded 58.42 KB, 1 files in 1s Download speed: 58.42 KB/s

... ...

Downloaded 649.92 MB, 2 files in 50s Download speed: 6.63 MB/s
Downloaded 646.92 MB, 2 files in 43s Download speed: 11.38 MB/s
****
Finished downloading version 1 of segmentation_hippocampus.
Downloaded files are in /var/tmp/test_models/hippocampus.
```

2.3. Running the Jupyter notebook examples

For each of the tasks, 3D segmentation and 2D classification, an example tutorial is provided in the format of the Jupyter notebook. To access notebook from browser you must forward 8888 port. To run examples, you also need to create a local folder and mount it (see each tutorial for the actual proper local folder and mount point names).

The directories containing Jupyter notebook files are:

- For 3D segmentation: **`/opt/nvidia/medical/segmentation/examples/brats`**
- For 2D classification: **`/opt/nvidia/medical/classification/examples/PLCO`**

You can replace the path, **`/opt/nvidia/medical/segmentation/examples/brats`**, in the example command below with the one of the Jupyter notebook directories.

Use this command:

```
docker run --runtime=nvidia -it --rm -v /your_local_path/brdemo:/mnt/brdemo -p
 8888:8888 -w /opt/nvidia/medical/segmentation/examples/brats nvcr.io/nvtltea/
med/tlt-annotate:v0.1-py3 jupyter notebook /opt/nvidia/medical/segmentation/
examples/brats --ip 0.0.0.0 --allow-root --no-browser
```

Here, instead of starting a docker in console, **`jupyter notebook`** is started. Additional parameters **`--ip 0.0.0.0 --allow-root --no-browser`** are allowed to access the notebook from outside the docker container. After running this command, look at the console response, copy and paste the link produced into a browser to access the notebook. Open the notebook to begin the demo.

Chapter 3.
USING TRANSFER LEARNING

3.1. End-to-end workflow of brain tumor segmentation

Running the notebook

Use the **`tutorial_brats.ipynb`** notebook to run this demo.

Use this command to run the docker:

```
docker run --runtime=nvidia -it --rm -v /your_local_path/brdemo:/mnt/brdemo -
p 8888:8888 -w /opt/nvidia/medical/segmentation/examples/brats $DOCKER_IMAGE
 jupyter notebook --ip 0.0.0.0 --allow-root --no-browser
```

If you are on a network that uses a proxy server to connect to the Internet, please see Running the container for information on how to include proxy server settings.

Where $DOCKER_IMAGE is the docker image name for example, nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3.

Open the tutorial_brats.ipynb notebook in your browser and follow the instructions in the notebook.

Your local folder , **`/your_local_path/brdemo`**, contains BraTS2018 data, pretrained models, and stores the output files.

Alternatively, you can also use the following instructions to reproduce the steps in the notebook manually.

Tutorial 1: Training from scratch (based on BraTS 2018 data)

This tutorial uses the BraTS 2018 challenge dataset . It is in NIfTI format with isotropic 1x1x1 mm voxel spacing.

- Input: 4 channel MRIs (T1c, T1, T2, FLAIR)

- Output: 3 channels of tumor subregions (tumor core (TC), whole tumor (WT), enhancing tumor(ET))

Start the docker in console/interactive mode

```
docker run --runtime=nvidia -it --rm -v /your_local_path/brdemo:/mnt/brdemo -p
 8888:8888 -w  /opt/nvidia/medical/segmentation/examples/brats $DOCKER_IMAGE /
bin/bash
```

Download data

Download the Brats2018 data, and place it in the **`brdemo/brats2018challenge`** folder. For details see https://www.med.upenn.edu/sbia/brats2018/registration.html

Run training

```
tlt-train segmentation -e config_brats_8.json \
                       -d config_brats18_datalist1.json \
                       -r /mnt/brdemo/output
```

Here **`config_brats_8.json`** is the main configuration file . You can also try the **`config_brats_16.json`** config file, if your GPU has >=16GB of memory. It is a better model with more filters, but it requires more memory. The provided datalist file, **`config_brats18_datalist1.json`**, lists all the data files used for training and validation. Finally /mnt/brdemo/output - is an output directory for model checkpoints and Tensorboard logs.

Convert the checkpoint to TensorRT optimized model

After trainining is done, we use `tlt-export` command to convert the checkpoint to TensorRT optimized model.

```
tlt-export --model_name 'model' --model_file_path /mnt/brdemo/output
 --input_node_names 'NV_INPUTALL_ITERATORGETNEXT' --output_node_names
 'NV_OUTPUT_LABEL'
```

This operations converts the TF model files (model.ckpt.data, model.ckpt.meta, model.ckpt.index inside /mnt/brdemo/output) to a compact optimized representation of TensorRT (.trt) files.

Inside of the /mnt/brdemo/output there is also a model_final.ckpt* files, which is TF checkpoint saved when you cancel training early in single GPU setting (as well as after the very last iteration). You can use --model_name 'model_final' instead of 'model'.

Evaluation

You can evaluate the accuracy of your TensorRT optimized model which is stored in **`/mnt/brdemo/output`**

(here evaluation computes accuracy metrics on the **`validation`** data of the **`config_brats18_datalist1.json`** file, it is the same information as computed after each epoch during training)

```
tlt-evaluate \
    --model_save_path /mnt/brdemo/output/model.trt.pb \
    --model_format trtmodel \
    --config config_validation.json \
    --file_root /mnt/brdemo/brats2018challenge \
    --data_file config_brats18_datalist1.json \
    --output_path /mnt/brdemo/output \
    --setup scanning_window \
    --no_overwrite
```

Inference

Run the model on some new data without the ground truth, and save the output segmentation masks.

```
tlt-infer \
    --model_save_path /mnt/brdemo/output/model.trt.pb \
    --model_format trtmodel \
    --config config_validation.json \
    --file_root /mnt/brdemo/brats2018challenge \
    --data_file config_brats18_datalist1.json \
    --output_path /mnt/brdemo/output \
    --setup scanning_window \
    --no_overwrite
```

This will save the segmentation masks (as NIfTI files) for all input files listed in **validation** provided in **`config_brats18_datalist1.json`**.

You can open the output_path folder and inspect the final segmentation masks in your 3D viewer (e.g. ITK-SNAP).

Tutorial 2: Fine-tuning a pre-trained model on a set of new images

Fine-tuning is similar to training, except the model is initialized from the pretrained weights. This tutorial has 1-channel input (T1c) and 1-channel output (TC).

Download the pre-trained model

Download the **`segmentation_brain_br16_t1c2tc_v1`** model and place into **`brdemo/pretrained/segmentation_brain_br16_t1c2tc_v1`** folder.

```
tlt-pull -k $API_KEY -m segmentation_brain_br16_t1c2tc -v 1 -o nvtltea -t med -
d /mnt/brdemo/pretrained/segmentation_brain_br16_t1c2tc
```

To view the list of available models use this command:

```
tlt-pull -lm -k $API_KEY -o nvtltea -t med
```

To show the versions of the model use this command:

```
tlt-pull -k $API_KEY -lv -m segmentation_brain_br16_t1c2tc -o nvtltea -t med
```

Fine-tuning or training

Fine-tuning is similar to training from scratch, except you initialize the model from the pre-trained weights (pre-trained checkpoint).

```
tlt-train segmentation -c /mnt/brdemo/pretrained/
segmentation_brain_br16_t1c2tc/model.ckpt -e config_brats_t1c2tc_f16_ft.json -d
 config_brats18_datalist_t1c.json -r /mnt/brdemo/output2
```

When fine-tuning is complete, the new checkpoint is saved to the **/mnt/brdemo/output2** directory.

Convert the checkpoint to TensorRT optimized model

After re-trainining is done, use the **tlt-export** command to convert the checkpoint to TensorRT optimized model.

```
tlt-export --model_name 'model' --model_file_path /mnt/brdemo/output2
 --input_node_names 'NV_INPUTALL_ITERATORGETNEXT' --output_node_names
 'NV_OUTPUT_LABEL'
```

If you did not re-train the model, you can try to convert the provided pre-trained checkpoint model instead.

```
cp -R /mnt/brdemo/pretrained/segmentation_brain_br16_t1c2tc /mnt/brdemo/output2
tlt-export --model_name 'model' --model_file_path /mnt/brdemo/output2
 --input_node_names 'NV_INPUTALL_ITERATORGETNEXT' --output_node_names
 'NV_OUTPUT_LABEL'
```

Evaluation

When the fine-tuning is complete, you can evaluate the accuracy of the newly trained model in **/mnt/brdemo/output2**.

Evaluation computes the accuracy metrics on the **validation** data of **config_brats18_datalist_t1c.json**, it is the same information as computed after each epoch during training and or fine-tuning, but more detailed.

```
tlt-evaluate \
    --model_save_path /mnt/brdemo/output2/model.trt.pb \
    --config config_validation_ft.json \
    --model_format trtmodel\
    --file_root /mnt/brdemo/brats2018challenge/ \
    --output_path /mnt/brdemo/output2 \
    --data_file config_brats18_datalist_t1c.json \
    --setup scanning_window \
    --no_overwrite
```

Aside from the console output of accuracy metrics per image, the output report will be saved to **/mnt/brdemo/output2/brats_dice_report.txt**, showing different metrics per file, and average metrics summary.

Inference

Inference is running the model on new data,without ground truth, and saving the output segmentation masks.

```
tlt-infer \
    --model_save_path /mnt/brdemo/output2/model.trt.pb \
    --config config_validation_ft.json \
    --model_format trtmodel \
    --file_root /mnt/brdemo/brats2018challenge/ \
    --output_path /mnt/brdemo/output2 \
    --data_file config_brats18_datalist_t1c.json \
    --setup scanning_window \
    --no_overwrite
```

This saves the segmentation masks as NIfTI files for all input files listed in **`validation`** provided in**`config_brats18_datalist_t1c.json`**. You can open the **`inference_output`** folder and inspect the final segmentation masks in a 3D viewer.

In this example the **`config_brats18_datalist_t1c.json`** and **`config_validation_ft.json`** files are reused for inference (which actually includes infor about the ground truth masks) - listings of masks can be removed since they are not used for inference.

3.2. End-to-end workflow of chest x-ray classification

This tutorial includes 4 demos:

- Training based on the Prostate, Lung, Colorectal and Ovarian (PLCO) chest x-ray data from an ImageNet pretrained model
- Fine tuning based on the PLCO chest x-ray data from pre-trained checkpoint model
- Converting the checkpoint model to TensorRT optimized model
- Performing inference and/or evaluation on the TensroRT optimized model

You should have the **`Models/ChestXrayClassification`** folder, and **`TestData/ChestXrayClassification`** folder with PLCO data downloaded and prepared see, Prepare the data. If using docker, you should have them mounted. The following code will start the Jupyter notebook server and you can run this **`tutorial_cxr.ipynb`** in **`medical/classification/examples/PLCO`**.

Here $DOCKER_IMAGE is the docker image name (e.g. nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3)

```
nvidia-docker run -it --network host --rm -v /path/to/Models/
ChestXrayClassification:/mnt/Models/ChestXrayClassification -v /path/to/
TestData/ChestXrayClassification:/mnt/TestData/ChestXrayClassification -p
 5000:5000 -p 8888:8888 $DOCKER_IMAGE /usr/local/bin/jupyter notebook /opt/
nvidia/medical/classification/examples/PLCO --ip=0.0.0.0 --allow-root --no-
browser
```

If you are on a network that uses a proxy server to connect to the Internet, please see Running the container for information on how to include proxy server settings.

Demo 1: Fine tuning on PLCO data using ImageNet pre-trained model

Define paths to the config files and data

- Input: 16-bit CXR pngs
- Output: Predictions of 15 disease abnormalities:
  - Nodule
  - Mass
  - Distortion of PulmonaryArchitecture

- Pleural Based Mass
- Granuloma
- Fluid in Pleural Space
- Right Hilar Abnormality
- Left HilarAbnormality
- Major Atelectasis
- Infiltrate
- Scarring
- Pleural Fibrosis
- Bone/Soft Tissue Lesion
- CardiacAbnormality
- COPD

Define paths to configuration files and data. It's also possible to supply these paths directly to scripts, but we'll define them as variables for convenience.

```
# json config with a list of data files
env DATA_LIST=/mnt/TestData/ChestXrayClassification/plco.json
# output directory to save outputs (such as Tensorboard logs and checkpoints)
env TRAIN_OUTPUT_DIR=/mnt/Models/ChestXrayClassification/
# json config with training configuration
%env TRAIN_CONFIG=/opt/nvidia/medical/classification/configs/
config_classification.json
```

Data

Prostate, Lung, Colorectal and Ovarian (PLCO) Cancer Screening Trial dataset is used for the tutorial. For more information see: https://biometry.nci.nih.gov/cdas/plco/.

A JSON file **`$DATA_LIST`** lists the training and validation data. This is an example of a data JSON file:

```
less $DATA_LIST
```

Configuration

The JSON config file $CONFIG_FILE specifies the training options, including data I/O transforms, data augmentation transforms, model, loss, optimizer, and hyperparameters. This is an example of a training config file:

```
cat $TRAIN_CONFIG
```

The configuration is setup for 12GB GPUs. If you have a GPU with less memory, you can edit the batch_size in the config file to a lower number.

Fine-tuning

Use the **`tlt-train`** command with the data index file **`$DATA_LIST`** and the **`$CONFIG_FILE`**, to run classification training and save the outputs including the tensorboard log and the checkpoint in **`$OUTPUT_DIR`**:

```
tlt-train classification \
    -e $TRAIN_CONFIG \
    -d $DATA_LIST \
    -r $TRAIN_OUTPUT_DIR
```

The expected mean Area Under the Curve (AUC) should be ~0.85.

Demo 2: Fine tuning on PLCO data using the checkpoint model

- Input: 16-bit CXR pngs
- Output: Predictions of 15 disease abnormalities. See the abnormalities listed in Demo 1.

Define paths to config files and data

First, define the paths to configuration files and data. It's also possible to supply these paths directly to scripts, but we'll define them as variables for convenience.

```
# pre-trained checkpoint file path with training configuration
env CKPT_CONFIG=/mnt/Models/ChestXrayClassification/model.ckpt
```

Fine-tuning

Use the **`tlt-train`** command with the data index file **`$DATA_LIST`** and the **`$CONFIG_FILE`**, to run classification training and save the outputs including the tensorboard log and the checkpoint in **`$OUTPUT_DIR`**:

```
tlt-train classification \
    -e $TRAIN_CONFIG \
    -d $DATA_LIST \
    -r $TRAIN_OUTPUT_DIR \
    -c $CKPT_CONFIG
```

Demo 3: Convert the trained checkpoint model to TensorRT optimized model

When trainining is complete, use **`tlt-export`** command to convert the checkpoint model to TensorRT optimized model. Here it downloads the pre-trained checkpoint and do the conversion. User can also convert the fine-tuned model by replacing the model file path.

```
# checkpoint file path with training configuration
env MODEL_PATH=/mnt/Models/ChestXrayClassification/
tlt-pull -m classification_chestxray -v 1 -k $API_KEY -d  $MODEL_PATH -o nvtltea
 -t med
tlt-export --model_name=model \
       --model_file_path=$MODEL_PATH \
       --input_node_names=NV_INPUT_IMAGE \
       --output_node_names=NV_OUTPUT_LABEL \
       --model_file_format=CKPT
```

Demo 4: Run Inference on Trained Model

When the training is complete, use the **`tlt-infer`** command to run an inference using the TensorRT optimized model. Inference/evaluation requires its own config file, corresponding to the training configuration file. It is similar to this:

```
# Configuration for inference
env /opt/nvidia/medical/classification/configs/config_inference.json
# model_save_path is the converted trt model from checkpoint
```

```
env MODEL_SAVE_PATH=/mnt/Models/ChestXrayClassification/model.trt.pb
# here, we use the trtmodel model format
env MODEL_FORMAT=trtmodel
# root location of image files
env FILE_ROOT=/mnt/TestData/ChestXrayClassification/PLCO_256_original
# output location of inference file
env INFERENCE_OUTPUT_PATH=/mnt/TestData/ChestXrayClassification/output
# list of images/label pairs, same as in training
env DATA_LIST=/mnt/TestData/ChestXrayClassification/plco.json
# specify the metagraph to use and the input/output signature
env METAGRAPH_TAG=inference
env SIGNATURE_TAG=inference_tensors
mkdir $INFERENCE_OUTPUT_PATH
cat $INFERENCE_CONFIG
```

Now run inference on files in your $DATA_LIST validation section.

```
tlt-infer \
    --config=$INFERENCE_CONFIG \
    --model_format=$MODEL_FORMAT \
    --model_save_path=$MODEL_SAVE_PATH \
    --data_file=$DATA_LIST \
    --metagraph_tag=$METAGRAPH_TAG \
    --signature_tag=$SIGNATURE_TAG \
    --output_path=$INFERENCE_OUTPUT_PATH \
    --file_root=$FILE_ROOT
```

This command generates a csv file containing the predictions for each image within the validation set in **`$DATA_LIST`**. The final report can be found at **`${INFERENCE_OUTPUT_PATH}/preds.csv`**. It looks like:

```
less ${INFERENCE_OUTPUT_PATH}/preds_outputs.csv
```

Run Evaluation

You can run a validation of the model on your data using the inference command. The ground truth is located in **`$DATA_LIST`**. You need to change your config to the validation configuration. If you specify **`--no_overwrite`**, you will use the existing predictions in **`${INFERENCE_OUTPUT_FILE}/preds_outputs.csv`** to measure performance, otherwise the script will rerun the inference.

```
# Configuration for validation
env VALIDATION_CONFIG=/opt/nvidia/medical/classification/configs/
config_validation.json
# Comment out if you want to redo the computations producing the inference
 outputs
env NO_OVERWRITE=--no_overwrite
cat ${VALIDATION_CONFIG}
```

Run validation with the environment variables updated:

```
tlt-evaluate \
    --config=$VALIDATION_CONFIG \
    --model_format=$MODEL_FORMAT \
    --model_save_path=$MODEL_SAVE_PATH \
    --data_file=$DATA_LIST \
    --metagraph_tag=$METAGRAPH_TAG \
    --signature_tag=$SIGNATURE_TAG \
    --output_path=$INFERENCE_OUTPUT_PATH \
    --file_root=$FILE_ROOT \
    $NO_OVERWRITE
```

This produces a set of .txt files in **`$INFERENCE_OUTPUT_PATH`**, one for each metric, containing the mean AUC value. The example also measures the mean AUC across all disease patterns and saves a text file with the result.

Chapter 4. WORKING WITH SEGMENTATION MODELS

This chapter provides instructions on preparing your data, training models, exporting, evaluating and performing inference on the trained segmentation models using transfer learning.

4.1. Prepare the data

All input images and labels must be in NIfTI format. Each input image and its corresponding label mask must have the same image dimension. To visualize or save NIfTI images, you can use free viewers such as ITK-SNAP or MITK.

If your native data format is different from NIfTI or if you want to convert the image and label mask to isotropic resolution, you can use the provided Data Converter or some other software of your choice, such as ITK-SNAP or directly in Python.

4.1.1. Using the data converter

If the data format is DICOM or the resolution is not isotropic, one can use the provided data converter tool to convert the data to isotropic NIfTI format. Furthermore, many pre-trained models were trained on 1x1x1mm resolution images, and to use those pre-trained models as a starting point, please convert the data to 1x1x1mm NIfTI format (Notice: If the dataset is already in NIfTI format, but not with 1x1x1 mm spacing, the data conversion is still required for the dataset.).

The **`tlt-dataconvert`** command converts all dicom volumes in your/data/directory to NIfTI format and optionally re-samples them to the provided resolution. If the images to be converted are segmentation labels, an option -l needs to be added, and the resampler will use nearest neighbor interpolator (otherwise linear interpolator is used).

```
tlt-dataconvert -d your/data/directory -r 1 -s .dcm -e .nii.gz -o your/output/
directory
```

Supported options are:

Option	Description
-d	Input directory with subdirectories containing dicom images.
-r	Output image resolution. If not provided, dicom resolution will be preserved. If only a single value is provided, target resolution will be isotrophic (e.g. -r 1 for 1x1x1mm resolution)
-s	Input file format, can be .dcm, .nii, .nii.gz, .mha, .mhd.
-e	Output file format, can be .nii, .nii.gz, .mha, .mhd.
-o	Output directory.
-f	(Optional) Force overwriting exsisting files if output directory already exists.
-l	(Optional) Flag indicating that the data is LABEL/SEGMENTATION masks and the nearest neighbor interpolation is used for re-sampling.

If you need to convert both 3D volumetric images and their segmentation labels, put them into two different folders, and run the converter once for the images and once for the labels using the `-l` flag.

4.1.2. Folder structure

The layout of data files can be arbitrary, but the JSON file describing the data list must contain the relative paths to all data files.

```
|--dataset_root:
     |--datalist.json
     |--train
        |--im1.nii.gz
        |--lb1.nii.gz
        |--im2.nii.gz
        |--lb2.nii.gz
        |--im3.nii.gz
        |--lb3.nii.gz
        |--im4.nii.gz
        |--lb4.nii.gz
    |--val
        |--im1.nii.gz
        |--lb1.nii.gz
        |--im2.nii.gz
        |--lb2.nii.gz
```

For example, the **`datalist.json`** file looks similar to this. Here all paths are relative to **`datalist.json`** location.

```
{
    "training": [
        {
            "image" : "train/im1.nii.gz",
            "label" : "train/lb1.nii.gz"
        },
        {
            "image" : "train/im2.nii.gz",
            "label" : "train/lb2.nii.gz"
        },
        {
            "image" : "train/im3.nii.gz",
            "label" : "train/lb3.nii.gz"
        },       {
            "image" : "train/im4.nii.gz",
            "label" : "train/lb4.nii.gz"
        },
    ],
    "validation": [
        {
            "image" : "val/im1.nii.gz",
            "label" : "val/lb1.nii.gz"
        },
        {
            "image" : "val/im2.nii.gz",
            "label" : "val/lb2.nii.gz"
        },
    ]
}
```

The **`training`** and **`validation`** lists contain the images that shall be used in training and validation steps, respectively.

By default, all paths inside the datalist.json are assumed relative to the datalist.json file location. You can optionally specify the ROOT base path of the datasets by specifying it in the main config file (image_base_dir JSON key) or as a command line option (--file_root) to tlt-train command.

4.1.3. Datalist JSON file

The JSON file describing the data structure must include the **`training`** key with a list of items (each containing **`image`** and **`label`** keys).

The value for the **`image`** key can be a string containing the path to a single NIfTI file or a list of strings that are paths to NIfTI files. If there are several channels they are saved as separate files. Here is an example:

```
        {
            "image" : [
                        "train/im1_ch1.nii.gz",
                        "train/im1_ch2.nii.gz",
                        "train/im1_ch3.nii.gz",
                        "train/im1_ch4.nii.gz"
                    ]
         "label" : "train/lb1.nii.gz"
        },
```

If **`image`** includes several files, they will be concatenated as separate channels of the network input. These images must be already spatially aligned.

The value for the **`label`** key, must be a string acontaining the path to a single NIfTI file with dense segmentation masks. The **`label`** mask can define segmentation using indices. Each integer index is a separate class or a multichannel one-hot-encoded image, where each channel represents a separate class.

The **`validation`** key is optional. If provided, the corresponding images/labels will be used to compute the validation metrics at the end of each training epoch in this release or less/more frequent if specified in the main training config. The validation section does not need to include the **`label`** keys, if the **`datalist.json`** is used for inference to compute the output segmentation masks.

4.2. Training a segmentation model

Segmentation training

Use the **`tlt-train segmentation`** command to perform segmentation training on the model.

```
tlt-train segmentation -e $TRAIN_CONFIG -d $DATA_LIST -r $TRAIN_OUTPUT_DIR -c
 $CHECKPOINT --file_root $FILEROOT
```

$TRAIN_CONFIG	Path to a json configuration file with all model parameters, input/ouput shapes, for example, `medical/segmentation/examples/brats/config_brats_16.json`
$DATA_LIST	Path to a JSON file with a list of input data file's relative to this file location. See Prepare the data
$TRAIN_OUTPUT_DIR	Path/name of the output directory for intermediate and final checkpoints, as well as Tensorboard logs
$CHECKPOINT	Optional if provided, the model is initialized with the previously pre-trained weights, otherwise if the checkpoint is not provided the model is initialized with default initialization or pre-trained imageNet model weights. The checkpoint file can be your own previously trained checkpoint or one of the provided pre-trained models.

$FILEROOT	Optional root folder of data files. If provided, all paths inside of the $DATA_LIST are assumed relative to this location.

Usage example

$TRAIN_CONFIG - is the main the configuration file. An example of it is shown below:

```
{
    "description" : "Example of Brats segmentation. 3 outputs after sigmoid for
 3 nested classes (TC,WT,ET).",
    "image_base_dir" : "/mnt/brdemo/brats2018challenge/",
    "num_channels": 4,
    "num_label_channels": 3,
    "num_classes": 3,

    "batch_size": 1,
    "epochs": 200,
    "num_workers": 8,
    "network_input_size": [224, 224, 128],
    "num_training_epoch_per_valid": 1,

    "record_step" : "every_epoch",
    "use_scanning_window" : false,
    "continue_from_checkpoint_epoch": true,
    "multi_gpu": false,

    "net_config":
       {
      "name": "SegmResnet",
      "use_vae": false,
      "blocks_down": "1,2,2,4",
      "blocks_up": "1,1,1",
      "init_filters": 8,
      "use_groupnorm": true,
      "use_groupnormG": 8,
      "reg_weight": 1e-5,
      "dropout_prob": 0.2,
      "final_activation": "sigmoid"

       },

    "auxiliary_outputs":
       [
      {
          "name": "common.metrics.metrics.dice_metric_masked_output",
          "is_onehot_targets": true,
          "is_independent_predictions": true,
          "tags": ["dice", "dice_tc", "dice_wt", "dice_et"]
      }
       ],

   "train":
    {
        "loss":
        {
            "name": "losses.dice_loss",
            "squared_pred": true,
            "is_onehot_targets": true
        },
        "optimizer":
        {
            "name": "Adam",
            "lr": 1e-3
        },

        "lr_policy":
        {
            "name": "lr_policy.ReduceLR_Poly",
            "poly_power": 0.9

        },
        "pre_transforms":
        [
            {
                "name": "transforms.LoadNifty",
                "fields": ["image", "label"]
            },
            {
                "name": "transforms.BratsConvertLabels",
                "fields": ["label"],
                "classes": ["TC", "WT", "ET"]
            },
            {
                "name": "transforms.CropSubVolumeRandomWithinBounds",
                "size": [224, 224, 128],
                "fields": ["image","label"]
            },
            {
                "name": "transforms.FlipAxisRandom",
                "fields": ["image","label"],
                "axis": [0,1,2]
            },
            {
                "name": "transforms.NormalizeNonzeroIntensities",
                "fields": "image"
            },
            {
                "name": "transforms.AugmentIntensityRandomScaleShift",
                "fields": "image"
            }
        ]

    },

  "validate":
    {
        "pre_transforms":
        [
            {
                "name": "transforms.LoadNifty",
                "fields": ["image", "label"]
            },
            {
                "name": "transforms.BratsConvertLabels",
                "fields": ["label"],
                "classes": ["TC", "WT", "ET"]
            },
            {
                "name": "transforms.CropSubVolumeCenter",
                "size": [224, 224, 128],
                "fields": ["image","label"]
            },
            {
                "name": "transforms.NormalizeNonzeroIntensities",
                "fields": "image"
            }
        ],

        "metrics":
```

```
        [
            {
                "name": "MetricAverage",
                "tag" : "mean_dice",
                "stopping_metric": true,
                "applied_key": "val_dice"
            },
            {
                "name": "MetricAverage",
                "tag" : "mean_dice_tc",
                "applied_key": "val_dice_tc"
            },
            {
                "name": "MetricAverage",
                "tag" : "mean_dice_wt",
                "applied_key": "val_dice_wt"
            },
            {
                "name": "MetricAverage",
                "tag" : "mean_dice_et",
                "applied_key": "val_dice_et"
            }

        ]

    }

}
```

Some details on individual fields are described here:

Field Key	**Description**
num_channels	Number of input channels
num_label_channels	Number of label channels
num_classes	Number of ground truth label classes
epochs	Number of training epochs
num_training_epoch_per_valid	Run validation every assigned number of training epochs
record_step	How often to save the checkpoint
use_scanning_window	Whether to use scanning window for validation
continue_from_checkpoint_epoch	Whether to continue from previous training
lr_policy	How to update learning rate
multi_gpu	Whether to train with multi-gpu using Horovod. Actual number of GPUs used depends on the parameters given to **`mpirun`**
num_workers	Number of cpu workers used to load and transform data (the higher the better, but

Field Key	Description
	should not exceed the number of cpu cores). Note if you are doing multi-gpu training then the num_workers should be total_num_cores/ num_gpus.
network_input_size	Input patch size to the network
data_format	Whether the network operates in channel-last or channel-first format.
net_config	Specification of the network used.
net_config:pretrain_weight_name	Path to pre-trained weights. If none, then network is randomly initialized.
train:loss	Loss function to use, For example, dice_loss.
train:optimizer	Optimizer configuration
train:transforms	The data loading and pre-processing transforms to execute for every image during training. For more information see, Data transforms and augmentations
validate:transforms	The data loading and pre-processing transforms to execute for every image during validation. Often different from training pre-processing. For more information see, Data transforms and augmentations
validate:metrics	Specifies the metrics to calculate in every validation epoch.
auxiliary_outputs	Extra outputs to compute for graph computation during training AND validation. These are often additional outputs that you want to track besides the loss. In training and validation the **`train_`** and **`val_`** strings, respectively, will be prepended to the output name.
image_base_dir	Optional root folder of data files. If provided, all paths inside of the $DATA_LIST are assumed relative to this location. Can be also specified by --file_root command line argument.

4.3. Multi-GPU training

To run multi-gpu training, wrap the tlt-train command inside of the mpirun, as shown below:

```
mpirun -np $NUM_GPU -H localhost:$NUM_GPU -bind-to none -map-by slot -x
 NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH -mca pml ob1 -mca btl ^openib --
allow-run-as-root \
tlt-train segmentation -e $TRAIN_CONFIG -d $DATA_LIST -r $TRAIN_OUTPUT_DIR -c
 $CHECKPOINT --file_root $FILEROOT |& grep -v "Read -1"
```

NUM_GPU is the number of GPUs to be allocated for training.

The field multi_gpu in the training configuration file (TRAIN_CONFIG) must be set to TRUE.

When training or finetuning the models in multi-GPU setting on small number of training data, it is recommended to adjust the learning rate provided in the configuration files, e.g. multiple the learning rate by the GPU number as is recommended in https://arxiv.org/pdf/1706.02677.pdf.

4.4. Tensorboard visualization

You can run the following command to use Tensorboard for visualization.

```
python3 -m tensorboard.main --logdir "${MODEL_DIR}"
```

4.5. Exporting the model to a TensorRT optimized model inference

After the model has been trained, use the **`tlt-export`** command to export the model for evaluation in either validation or inference.

This command accepts the Checkpoint format as input, and produces a regular frozen graph and a TensorRT optimized graph file. The Checkpoint model format contains a graph meta file, a data file, and an index file.

Command Syntax

The syntax of the **`tlt-export`** command is:

`tlt-export` ***`options`***

Options	Description
`--model_file_format <modelFileFormat>`	Specifies the file format of source model.

Options	Description
	CKPT - the source model is in Checkpoint format
--model_file_path <modelFileDir>	The file directory that contains the model files.
--model_name <modelName>	The base name of the model files. It could be **model** or **final_model**. Check the file names in the model file path.
--input_node_names <inputNodeNames>	The input node names of the model. Depending on the model, the input node name is similar to **NV_INPUT_IMAGE** or **cond/Merge**.
--output_node_names <outputNodeNames>	The output node names of the model. The value is **NV_OUTPUT_LABEL**.
--checkpoint_ext <checkpointExtension>	The value is **.ckpt**.
--meta_file_ext <metaFileExtension>	The value is **.meta**.
--regular_frozen_file_ext <regularFrozenFileExtension>	The value is **.fzn.pb**.
--trt_file_ext <trtOptimizedFileExtension>	The value is **.trt.pb**.
--trt_precision_mode <trtPrecisionMode>	Precision model for TRT optimization. The value can be either **FP16** or **FP32**. Default value is **FP32**.
--trt_dynamic_mode	If specified, TRT-optimization will be done in dynamic mode. There seems to be some issue with TRT optimization in dynamic mode, especially if minimum segment size is small. It is advised that the minimum segment size should be > 3 for this to work properly.
--max_batch_size <maxBatchSize>	The maximum batch size for TRT optimized graph. Inference won't work if the actual batch size during inference is

Options	Description
	greater than the specified size, The default size is 5.
`--trt_min_seg_size <minimumSegmentSize>`	The minimum segment size for TRT optimization. TRT optimization is applied to a subgraph only if the subgraph's size is greater than the specified size. Setting this to a large value may help avoid the issue with TRT optimization in dynamic mode.

Command Output

This command generates a regular frozen graph file with the extension **`.fzn.pb`** and a TRT-optimized graph file with the extension **`.trt.pb`**. Both files are placed in the specified model file directory.

Both graph files can be used for model evaluation, but the intent is to only use TRT graph. Here are some examples:

- To export a checkpoint format with Static TRT optimization:

  ```
  tlt-export --model_file_format CKPT \
  --model_file_path yourModelFileDirectory \
  --model_name model  \
  --input_node_names NV_INPUT_IMAGE \
  --output_node_names NV_OUTPUT_LABEL \
  --trt_min_seg_size 5
  ```

  This produces **`model.fzn.pb`** and **`model.trt.pb`** in **`yourModelFileDirectory`**.
- To export a checkpoint format with dynamic TRT optimization:

  ```
  tlt-export --model_file_format CKPT \
  --model_file_path yourModelFileDirectory \
  --model_name model  \
  --input_node_names NV_INPUT_IMAGE \
  --output_node_names NV_OUTPUT_LABEL \
  --trt_min_seg_size 5 --trt_dynamic_mode
  ```

4.6. Segmentation model evaluation with ground truth

Use **`tlt-evaluate`** to run evaluation of a list of images based on a set of ground truth labels:

```
tlt-evaluate \
    --model_save_path $MODEL_SAVE_PATH \
    --config $EVALUATION_CONFIG \
    --model_format $MODEL_FORMAT \
    --file_root $FILE_ROOT \
```

```
    --output_path $OUTPUT_PATH \
    --data_file $JSON_FILE \
    --setup scanning_window \
    $OVERWRITE
```

$EVALUATION_CONFIG	Path to a JSON configuration file with details on loading and transforms. The only difference from the inference configuration is the presence of metrics and label_transforms, if any. Should match the training_config in these respects, for example **`/medical/segmentation/examples/brats/config_validation.json`**.
$MODEL_FORMAT	**`trtmodel`**.
$MODEL_SAVE_PATH	Path to the model you are performing evaluation on. It should be to a **`.pb`** file.
$JSON_FILE	Path to a JSON file with a list of input data files relative to the file's location. See Prepare the data.
$SETUP	Either, **`scanning window`** which is used when the image is larger than the input size of network, or **`standard`** which is used when the image has the same size as the input size of the network.
$OUTPUT_PATH	Folder path where the evaluation output will be saved.
$FILE_ROOT	Root folder of images.
$NO_OVERWRITE	Export NO_OVERWRITE to **`–no_overwrite`**, if you don't want to overwrite any existing results.

$EVALUATION_CONFIG - is the main configuration file. And example of it is shown below:

```
{
    "input_nodes":
    {
        "image": "NV_INPUTALL_ITERATORGETNEXT"
    },
    "output_nodes":
    {
        "model": "NV_OUTPUT_LABEL"
    },
    "batch_size": 1,
    "pre_transforms":
    [
        {
            "name": "transforms.LoadResolutionFromNifty",
            "applied_key": "image",
            "new_key": "image_resolution"
        },
        {
            "name": "transforms.LoadNifty",
            "fields": "image"
        },
        {
            "name": "transforms.NormalizeNonzeroIntensities",
            "fields": "image"
        },
        {
            "name": "transforms.ResampleVolume",
            "applied_key": "image",
            "resolution": "image_resolution",
            "target_resolution": [1.0,1.0,1.0]
        }
    ],
    "post_transforms":
    [
        {
            "name": "transforms.ThresholdValues",
            "applied_key": "model",
            "threshold": 0.5
        },
        {
            "name": "transforms.ResampleVolume",
            "applied_key": "model",
            "resolution": [1.0,1.0,1.0],
            "target_resolution": "image_resolution"
        },
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "model",
            "channel_names": ["TC", "WT", "ET"]
        }
    ],
    "label_transforms":
    [
        {
            "name": "transforms.LoadNifty",
            "fields": "label"
        },
        {
            "name": "transforms.BratsConvertLabels",
            "fields": ["label"],
            "classes": ["TC", "WT", "ET"]
        },
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "label",
            "channel_names": ["TC_label", "WT_label", "ET_label"]
        }
    ],
    "writers":
    [
        {
            "applied_key": "TC",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "WT",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "ET",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        }
    ],
    "val_metrics":
    [
        {
            "name": "MetricAverageFromArrayDice",
            "tag" : "mean_dice_tc",
            "applied_key": "TC",
            "label_key": "TC_label"
        },
        {
            "name": "MetricAverageFromArrayDice",
            "tag" : "mean_dice_wt",
            "applied_key": "WT",
            "label_key": "WT_label"
        },
        {
            "name": "MetricAverageFromArrayDice",
            "tag" : "mean_dice_et",
            "applied_key": "ET",
            "label_key": "ET_label"
        }
    ]
}
```

The only difference from the inference config, for this example, is the presence of validation metrics, which specify the set of metrics you want to compute on the evaluation set. In the above example, we are measuring mean Dice's score of all the disease patterns, plus the mean Dice's score across all segmentation classes. This will save a text file for each metric.

Here are some details on individual fields:

Field Key	**Description**
input_nodes	A graph node name in the graph for input.
output_nodes	A graph node name in the graph for output.
pre-transforms	Chain of transforms to execute for loading and pre-processing the data. Should match that found in the training configuration. For more information see, Data transforms and augmentations
writers	List of writers that will output the evaluation results to the file system. No segmentation masks will be generated in evaluation task.
val_metrics	List of metric classes that specifying what metrics to measure on the evaluation set.

4.7. Segmentation model inference

Use the **`tlt-infer`** command to run inference on the model.

```
tlt-infer \
    --model_save_path $MODEL_SAVE_PATH \
    --config =$INFERENCE_CONFIG \
    --model_format $MODEL_FORMAT \
    --file_root $FILE_ROOT \
    --output_path $OUTPUT_PATH \
    --data_file $JSON_FILE \
    --setup scanning_window \
    $OVERWRITE
```

$INFERENCE_CONFIG	Path to a JSON configuration file with details on loading and transforms. The only difference between the validaton configuration and the inference configuration is the presence of metrics and label_transforms, if any. It should match the training_config file in these respects, e.g. **`/medical/segmentation/tasks/brats/config_validation.json`**.
$MODEL_FORMAT	**`trtmodel`**
$MODEL_SAVE_PATH	Path to the model you are performing an inference on. It should be to a .pb file.
$JSON_FILE	Path to a JSON file with a list of input data files relative to this file's location. See Segmentation Data Preparation.
$SETUP	Either, scanning window which is used when the input image is larger than the input size of the model, or 'standard' which is used when the input image has the same size as input of the model.
$OUTPUT_PATH	The folder path where the inference output (segmentation masks) are saved.
$FILE_ROOT	Root folder of images.
$NO_OVERWRITE	Export NO_OVERWRITE to **`–no_overwrite`**, if you don't want to overwrite any existing results, for example, if you had ran the same inference loop on another occasion.

$INFERENCE_CONFIG - is main configuration file. Here is an example:

```
{
    "input_nodes":
    {
        "image": "NV_INPUTALL_ITERATORGETNEXT"
    },
    "output_nodes":
    {
        "model": "NV_OUTPUT_LABEL"
    },
    "batch_size": 1,
    "pre_transforms":
    [
        {
            "name": "transforms.LoadResolutionFromNifty",
            "applied_key": "image",
            "new_key": "image_resolution"
        },
        {
            "name": "transforms.LoadNifty",
            "fields": "image"
        },
        {
            "name": "transforms.NormalizeNonzeroIntensities",
            "fields": "image"
        },
        {
            "name": "transforms.ResampleVolume",
            "applied_key": "image",
            "resolution": "image_resolution",
            "target_resolution": [1.0,1.0,1.0]
        }

    ],

    "post_transforms":
    [
        {
            "name": "transforms.ThresholdValues",
            "applied_key": "model",
            "threshold": 0.5
        },
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "model",
            "channel_names": ["TC", "WT", "ET"]
        },
        {
            "name": "transforms.ResampleVolume",
            "applied_key": "image",
            "resolution": [1.0,1.0,1.0],
            "target_resolution": "image_resolution"
        }

    ],

    "writers":
    [
        {
            "applied_key": "TC",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "WT",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "ET",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        }
    ]
}
```

You can use the same configuration file for evaluation here as well. In this scenario, the metric values specified in the configuration file won't be computed. For inference, no ground truth label is needed.

Here are details on the fields of the configuration file used:

Field	Description
input_nodes	A graph node name in the graph for input.
output_nodes	A graph node name in the graph for output.
pre_transforms	Chain of transforms to execute for loading and pre-processing the data. Should partially match that found in the training configuration. For more information see, Data transforms and augmentations
writers	List of writers that will output the inference results to the file system. In this case, we have three writers that will output segmentation masks for three classes.

Chapter 5. WORKING WITH CLASSIFICATION MODELS

The chapter provides instructions on preparing your data, training models, exporting, evaluating and performing inference on the trained 2D classification models with transfer learning.

5.1. Prepare the data

This section describes the format in which the data can be used with transfer learning for 2D classification tasks.

5.1.1. Data format

All input images and labels must be in png format. If you are planning to resample images, e.g., to 256x256, it is best to do that as a pre-processing step, rather than have the TLT toolkit do that on the fly. The png files can be 8- or 16-bit. You must also have ground truth labels available. These are often binary, i.e., {0,1}, or multi-class, i.e., {0,C} if there are C classes.

5.1.2. Folder structure

The layout of data files can be arbitrary, but the JSON file describing the data list must contain relative paths to all image files.

```
|--dataset_root:
     |--datalist.json
     |--png_files
        |--im1.png
        |--im2.png
        |--im3.png
```

5.1.3. Datalist JSON file

The JSON file describing the data structure must include a **`label_format`** key. The corresponding value should be a list of natural numbers, specifying the number of type of labels in the dataset. For instance, for the PLCO dataset, there are 15 binary labels, so it should be a list of 15 ones: [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]. If you have multi-class labels, then the 1's should be replaced by the number of classes in each label.

The datafile should also have a **`training`** and **`validation`** key. These keys contain:

- a list of dictionaries, where the value for the **`image`** key must be a relative path to the png file.
- the value for the **`label`** key must be a list of natural numbers corresponding to the ground truth labels.

The labels for each image must match the **`label_format`** specified above.

```
{
    "label_format": [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    "training":
    [
        {
            "image" : "im1.png"
            "label" : [0,0,1,0,0,0,0,0,0,0,1,0,0,0,0]
        },
...
```

The **`validation`** key is optional and only should be specified if the main training config file specifies metrics to compute. If the **`validation`** key is provided, it specifies the corresponding images and labels used to compute the validation metrics at the end of each training epoch (or less/more frequent if specified in the main training config).

5.2. Training a classification model

Use the **`tlt-train classification`** to run the training:

```
tlt-train classification -e $TRAIN_CONFIG -d $DATA_LIST -r $TRAIN_OUTPUT_DIR -c
 $CHECKPOINT --file_root $FILEROOT
```

$TRAIN_CONFIG	Path to a JSON configuration file with all model parameters, input/ ouput shapes, for example **`/medical/classification/configs/config_classification.json`**
$DATA_LIST	Path to a JSON file with a list of input data files relative to this file's location.
$TRAIN_OUTPUT_DIR	Path or name of the output directory for intermediate and final checkpoints, as well as Tensorboard logs.
$CHECKPOINT	This is optional. If provided, the model is initialized with a checkpoint from a prior run, otherwise the model is either randomly initialized or from ImageNet pre-trained weights.

$FILEROOT	Optional root folder of data files. If provided, all paths inside of the $DATA_LIST are assumed relative to this location.

$TRAIN_CONFIG - is the main configuration file. Here is an example of it:

```
{
    "image_base_dir": "/Data/CXR/PLCO/PLCO_256_original",
    "weight_decay": 1e-5,
    "batch_size": 40,
    "epochs": 40,
    "multi_gpu": true,
    "num_workers": 8,
    "network_input_size": [256, 256],
    "data_format": "channels_last",
    "net_config":
    {
        "name": "Densenet121",
        "pretrain_weight_name": "/Data/pretrained_models/imagenet/densenet/
densenet121/densenet121_weights_tf.h5"
    },
    "train":
    {
        "loss":
        {
            "name": "losses.classification_loss"
        },
        "optimizer":
        {
            "name": "Adam",
            "lr": 2e-4
        },
        "pre_transforms":
        [
            {
                "name": "transforms.LoadPng",
                "fields": ["image"]
            },
            {
                "name": "transforms.CropRandomSubImageInRange",
                "lower_size": [0.9,0.9],
                "data_format": "grayscale",
                "image_field": "image",
                "max_displacement": 200
            },
            {
                "name": "transforms.NPResizeImage",
                "applied_keys": ["image"],
                "output_shape": [256,256],
                "data_format": "grayscale"
            },
            {
                "name": "transforms.NP2DRotate",
                "applied_keys": ["image"],
                "angle": 7,
                "random": true,
                "data_format": "grayscale"
            },
            {
                "name": "transforms.NPExpandDims",
                "applied_keys": "image",
                "expand_axis": 2
            },
            {
                "name": "transforms.NPRepChannels",
                "applied_keys": "image",
                "channel_axis": 2,
                "repeat": 3
            },
            {
                "name": "transforms.CenterData",
                "applied_keys": "image",
                "subtrahend": [2876.37, 2876.37,2876.37],
                "divisor": [883, 883, 883]
            }
        ]
    },
    "validate":
    {
        "pre_transforms":
        [
            {
                "name": "transforms.LoadPng",
                "fields": ["image"]
            },
            {
                "name": "transforms.NPResizeImage",
                "applied_keys": ["image"],
                "output_shape": [256,256],
                "data_format": "grayscale"
            },
            {
                "name": "transforms.NPExpandDims",
                "applied_keys": "image",
                "expand_axis": 2
            },
            {
                "name": "transforms.NPRepChannels",
                "applied_keys": "image",
                "channel_axis": 2,
                "repeat": 3
            },
            {
                "name": "transforms.CenterData",
                "applied_keys": "image",
                "subtrahend": [2876.37, 2876.37,2876.37],
                "divisor": [883, 883, 883]
            }
        ],
        "metrics":
        [
            {
                "name": "MetricAverage",
                "tag" : "mean_accuracy",
                "applied_key": "val_accuracy"
            },
            {
                "name": "MetricAUC",
                "tag" : "Average_AUC",
                "applied_key": "binary_preds",
                "label_key": "binary_labels",
                "auc_average": "macro",
                "stopping_metric": true
            },
            {
                "name": "MetricAUC",
                "tag" : "Nodule",
                "class_index": 0,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Mass",
                "class_index": 1,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
                    {
                "name": "MetricAUC",
                "tag" : "Distortion_pulmonary_architecture",
                "class_index": 2,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Pleural_based_mass",
                "class_index": 3,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Granuloma",
                "class_index": 4,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Fluid_in_pleural_space",
                "class_index": 5,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Right_hilar_abnormality",
                "class_index": 6,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Left_hilar_abnormality",
                "class_index": 7,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Major_atelectasis",
                "class_index": 8,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Infiltrate",
                "class_index": 9,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Scarring",
                "class_index": 10,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Pleural_fibrosis",
                "class_index": 11,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Bone_soft_tissue_lesion",
                "class_index": 12,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "Cardiac_abnormality",
                "class_index": 13,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            },
            {
                "name": "MetricAUC",
                "tag" : "COPD",
                "class_index": 14,
                "applied_key": "binary_preds",
                "label_key": "binary_labels"
            }
        ]
    },
    "auxiliary_outputs":
    [
        {
            "name": "common.metrics.metrics.compute_accuracy",
            "tags": "accuracy",
            "use_sigmoid": true
        }
    ]
}
```

Here are some details on individual fields:

Field Key	**Description**
image_base_dir	Root location of the images
weight_decay	The L2 regularization weight
multi_gpu	Whether to train with multi-gpu using Horovod. Actual number of GPUs used depends on the parameters given to \`mpirun\`
num_workers	Number of cpu workers used to load and transform data (the higher the better, but should not exceed the number of cpu cores). Note if you are doing multi-gpu training then the num_workers should be total_num_cores/num_gpus

Field Key	Description
network_input_size	Input size to network
data_format	Whether the network operates in channels last or channels first format.
net_config	Dictionary specifying the network used
net_config:pretrain_weight_name	Path to pre-trained weights. If none, then network is randomly initialized
train:loss	Loss function to use. Should always be classification_loss, which can handle binary or multi-class multi-labels
train:optimizer	Optimizer configuration
train:pre_transforms	The data loading and pre-processing transforms to execute for every image during training. For more information see, Data transforms and augmentations
validate:pre_transforms	The data loading and pre-processing transforms to execute for every image during validation. Often different from training pre-processing. For more information see, Data transforms and augmentations
validate:metrics	Specifies the metrics to calculate every validation epoch.
auxiliary_outputs	Extra outputs to compute for graph computation during training AND validation. These are often additional outputs that you want to track besides the loss. In training and validation the **`train_`** and **`val_`** strings, respectively, will be prepended to the output name.

5.3. Multi-GPU training

To run multi-gpu training, wrap the **`tlt-train classification command`** inside of the mpirun, as shown below:

```
mpirun -np ${NUM_GPU} -H localhost:$NUM_GPU -bind-to none -map-by slot -x
 NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH -mca pml ob1 -mca btl ^openib --
allow-run-as-root \
tlt-train classification -e $TRAIN_CONFIG -d $DATA_LIST -r $TRAIN_OUTPUT_DIR -c
 $CHECKPOINT --file_root $FILEROOT |& grep -v "Read -1"
```

NUM_GPU is the number of GPUs to be allocated for training.

The field **multi_gpu** in the training configuration file (TRAIN_CONFIG) must be set to TRUE.

When training or finetuning the models in multi-GPU setting on small number of training data, it is recommended to adjust the learning rate provided in the configuration files, e.g. multiple the learning rate by the GPU number as is recommended in https://arxiv.org/pdf/1706.02677.pdf.

5.4. Tensorboard visualization

You can run the following command to use Tensorboard for visualization.

```
python3 -m tensorboard.main --logdir "${MODEL_DIR}"
```

5.5. Exporting the model to a TensorRT optimized model inference

After the model has been trained, use the **tlt-export** command to export the model for evaluation in either validation or inference.

This command accepts the Checkpoint format as input, and produces a regular frozen graph and a TensorRT optimized graph file. The Checkpoint model format contains a graph meta file, a data file, and an index file.

Command Syntax

The syntax of the **tlt-export** command is:

tlt-export ***options***

Options	Description
--model_file_format <modelFileFormat>	Specifies the file format of source model. CKPT - the source model is in Checkpoint format
--model_file_path <modelFileDir>	The file directory that contains the model files.
--model_name <modelName>	The base name of the model files. It could be **model** or **final_model**. Check the file names in the model file path.
--input_node_names <inputNodeNames>	The input node names of the model. Depending on the model, the input node

Options	Description
	name is similar to **`NV_INPUT_IMAGE`** or **`cond/Merge`**.
`--output_node_names <outputNodeNames>`	The output node names of the model. The value is **`NV_OUTPUT_LABEL`**.
`--checkpoint_ext <checkpointExtension>`	The value is **`.ckpt`**.
`--meta_file_ext <metaFileExtension>`	The value is **`.meta`**.
`--regular_frozen_file_ext <regularFrozenFileExtension>`	The value is **`.fzn.pb`**.
`--trt_file_ext <trtOptimizedFileExtension>`	The value is **`.trt.pb`**.
`--trt_precision_mode <trtPrecisionMode>`	Precision model for TRT optimization. The value can be either **`FP16`** or **`FP32`**. Default value is **`FP32`**.
`--trt_dynamic_mode`	If specified, TRT-optimization will be done in dynamic mode. There seems to be some issue with TRT optimization in dynamic mode, especially if minimum segment size is small. It is advised that the minimum segment size should be > 3 for this to work properly.
`--max_batch_size <maxBatchSize>`	The maximum batch size for TRT optimized graph. Inference won't work if the actual batch size during inference is greater than the specified size, The default size is 5.
`--trt_min_seg_size <minimumSegmentSize>`	The minimum segment size for TRT optimization. TRT optimization is applied to a subgraph only if the subgraph's size is greater than the specified size. Setting this to a large value may help avoid the issue with TRT optimization in dynamic mode.

Command Output

This command generates a regular frozen graph file with the extension **.fzn.pb** and a TRT-optimized graph file with the extension **.trt.pb**. Both files are placed in the specified model file directory.

Both graph files can be used for model evaluation, but the intent is to only use TRT graph. Here are some examples:

- To export a checkpoint format with Static TRT optimization:

  ```
  tlt-export --model_file_format CKPT \
  --model_file_path yourModelFileDirectory \
  --model_name model  \
  --input_node_names NV_INPUT_IMAGE \
  --output_node_names NV_OUTPUT_LABEL \
  --trt_min_seg_size 5
  ```

  This produces **model.fzn.pb** and **model.trt.pb** in **yourModelFileDirectory**.
- To export a checkpoint format with dynamic TRT optimization:

  ```
  tlt-export --model_file_format CKPT \
  --model_file_path yourModelFileDirectory \
  --model_name model  \
  --input_node_names NV_INPUT_IMAGE \
  --output_node_names NV_OUTPUT_LABEL \
  --trt_min_seg_size 5 --trt_dynamic_mode
  ```

5.6. Classification model evaluation with ground truth

Use **tlt-evaluate** to run evaluation of a list of images, based on the corresponding set of ground truth labels:

```
tlt-evaluate \
    --config=$VALIDATION_CONFIG \
    --model_format=$MODEL_FORMAT \
    --model_save_path=$MODEL_SAVE_PATH \
    --data_file=$DATA_LIST \
    --metagraph_tag=$METAGRAPH_TAG \
    --signature_tag=$SIGNATURE_TAG \
    --output_path=$INFERENCE_OUTPUT_FILE \
    --file_root=$FILE_ROOT \
    $NO_OVERWRITE
```

Where:

$VALIDATION_CONFIG	Path to a JSON configuration file with details on loading and transforms. Only difference from the inference configuration is the presence of metrics and label_transforms, if any. Should match the training_config in these respects, e.g. /medical/classification/configs/config_inference.json

$MODEL_FORMAT	**`trtmodel`**
$MODEL_SAVE_PATH	Path to the model you are performing inference on. It should be to a \`.pb\` file
$DATA_LIST	Path to a JSON file with a list of input data files (relative to this file's location).
$METAGRAPH_TAG	**`inference`**
$SIGNATURE_TAG	**`inference_tensors`**
$INFERENCE_OUTPUT_FILE	Folder path where the inference output will be saved. For classification, this is just a .csv file of raw logit predictions for every image.
$FILE_ROOT	Root folder of images
$NO_OVERWRITE	Export NO_OVERWRITE to "–no_overwrite", if you don't want to overwrite any existing results, e.g., if you had ran the same inference loop on another occasion

$VALIDATION_CONFIG - is the main configuration file. Here is an example of it:

```
{

    "input_nodes":
    {
        "image": "inputs"
    },
    "output_nodes":
    {
        "model": "outputs"
    },

    "batch_size": 40,

    "pre_transforms":
    [
        {
            "name": "transforms.LoadPng",
            "fields": ["image"]
        },

        {
            "name": "transforms.NPResizeImage",
            "applied_keys": ["image"],
            "output_shape": [256,256],
            "data_format": "grayscale"
        },

        {
            "name": "transforms.NPExpandDims",
            "applied_keys": "image",
            "expand_axis": 2
        },
        {
            "name": "transforms.NPRepChannels",
            "applied_keys": "image",
            "channel_axis": 2,
            "repeat": 3
        },
        {
            "name": "transforms.CenterData",
            "applied_keys": "image",
            "subtrahend": [2876.37, 2876.37,2876.37],
            "divisor": [883, 883, 883]
        }


    ],

    "writers":
    [
        {
            "applied_key": "outputs",
            "name": "writers.WriteClassificationResults"
        }
    ]

    "val_metrics":
    [
        {
            "name": "MetricAUC",
            "tag" : "Average_AUC",
            "applied_key": "outputs",
            "label_key": "label",
            "auc_average": "macro"
        },
        {
            "name": "MetricAUC",
            "tag" : "Nodule",
            "class_index": 0,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Mass",
            "class_index": 1,
            "applied_key": "outputs",
            "label_key": "label"
        },
                {
            "name": "MetricAUC",
            "tag" : "Distortion_pulmonary_architecture",
            "class_index": 2,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Pleural_based_mass",
            "class_index": 3,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Granuloma",
            "class_index": 4,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Fluid_in_pleural_space",
            "class_index": 5,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Right_hilar_abnormality",
            "class_index": 6,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Left_hilar_abnormality",
            "class_index": 7,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Major_atelectasis",
            "class_index": 8,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Infiltrate",
            "class_index": 9,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Scarring",
            "class_index": 10,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Pleural_fibrosis",
            "class_index": 11,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Bone_soft_tissue_lesion",
            "class_index": 12,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "Cardiac_abnormality",
            "class_index": 13,
            "applied_key": "outputs",
            "label_key": "label"
        },
        {
            "name": "MetricAUC",
            "tag" : "COPD",
            "class_index": 14,
            "applied_key": "outputs",
            "label_key": "label"
            }
        ]
}
```

The only difference from the inference config, for this example, is the presence of validation metrics, which specify the set of metrics you want to compute on the validation set. In the above example, we are measuring mean AUC of all the disease patterns, plus the mean AUC across all disease patterns. This will save a text file for each metric with the AUC number inside.

Here are details of the fields used:

Field Key	Description
input nodes	Mapping from input tensors to a tag used in inference. If you are using a frozen model, then the mapping should specify a tensor name in the graph.
output_nodes	Mapping from output tensors to a tag used in inference. If you are using a frozen model, then the mapping should specify a tensor name in the graph.
pre-transforms	Chain of transforms to execute for loading and pre-processing the data. Should match that found in the training configuration.
writers	List of writers that will output the inference results to the file system. In this case, we have one writer that will output a .csv file of raw prediction results.
val_metrics	List of metric classes that specifying what metrics to measure on the validation set.

5.7. Classification model inference

Use the **`tlt-infer`** command to run inference on a model:

```
tlt-infer \
    --config=$INFERENCE_CONFIG \
    --model_format=$MODEL_FORMAT \
    --model_save_path=$MODEL_SAVE_PATH \
    --data_file=$DATA_LIST \
    --metagraph_tag=$METAGRAPH_TAG \
    --signature_tag=$SIGNATURE_TAG \
    --output_path=$INFERENCE_OUTPUT_PATH \
    --file_root=$DATA_ROOT \
    --$NO_OVERWRITE
```

Where:

$INFERENCE_CONFIG	Path to a JSON configuration file with details on loading and transforms. Should match the training_config in these respects, e.g. **`/medical/classification/configs/config_inference.json`**
$MODEL_FORMAT	**`trtmodel`**
$MODEL_SAVE_PATH	Path to the model you are performing inference on **`classification_chestxray`**.
$DATA_LIST	Path to a JSON file with a list of input data files (relative to this file location). See Prepare the data.
$METAGRAPH_TAG	If using inference, then the metagraph tag indicates which metagraph you want to load from the SavedModel. The default is **`inference`** .
$SIGNATURE_TAG	**`inference_tensors`**
$INFERENCE_OUTPUT_PATH	The folder path where the inference ouput will be saved. For classification, this is just a .csv file of raw logit predictions for every image.
$FILE_ROOT	Root folder of images.
$NO_OVERWRITE	Export NO_OVERWRITE to "–no_overwrite", if you don't want to overwrite any existing results, e.g., if you had ran the same inference loop on another occasion.

$EVALUATION_CONFIG - is the main configuration file. Here is an example of it:

```
{
    "input_nodes":
    {
        "image": "inputs"
    },
    "output_nodes":
    {
        "model": "outputs"
    },
    "batch_size": 40,
    "pre_transforms":
    [
        {
            "name": "transforms.LoadPng",
            "fields": ["image"]
        },
        {
            "name": "transforms.NPResizeImage",
            "applied_keys": ["image"],
            "output_shape": [256,256],
            "data_format": "grayscale"
        },
        {
            "name": "transforms.NPExpandDims",
            "applied_keys": "image",
            "expand_axis": 2
        },
        {
            "name": "transforms.NPRepChannels",
            "applied_keys": "image",
            "channel_axis": 2,
            "repeat": 3
        },
        {
            "name": "transforms.CenterData",
            "applied_keys": "image",
            "subtrahend": [2876.37, 2876.37,2876.37],
            "divisor": [883, 883, 883]
        }
    ],
    "writers":
    [
        {
            "applied_key": "outputs",
            "name": "writers.WriteClassificationResults"
        }
    ]
}
```

Here are details on the fields used:

Field Key	Description
input nodes	A graph node name in the graph for input.
output_nodes	A graph node name in the graph for output.
pre-transforms	Chain of transforms to execute for loading and pre-processing the data. Should match that found in the training configuration. For more information see, Data transforms and augmentations.
writers	List of writers that will output the inference results to the file system. In this case, we have one writer that will output a .csv file of raw prediction results.

Chapter 6. APPENDIX

6.1. Segmentation models

Here is a list of the segmentation models. All the model are trained using 1x1x1mm resolution data.

Segmentation model	Description
Brain tumor segmentation	
▸ segmentation_brain_br16_full	A pre-trained model for volumetric (3D) segmentation of brain tumors from multi-modal MRIs based on BraTS 2018 data. https://www.med.upenn.edu/sbia/brats2018/data.html The model is trained to segment 3 nested subregions of primary (gliomas) brain tumors: the "enhancing tumor" (ET), the "tumor core" (TC), the "whole tumor" (WT) based on 4 input MRI scans (T1c, T1, T2, FLAIR). The ET is described by areas that show hyper-intensity in T1c when compared to T1, but also when compared to "healthy" white matter in T1c. The TC describes the bulk of the tumor, which is what is typically resected. The TC entails the ET, as well as the necrotic (fluid-filled) and the non-enhancing (solid) parts of the tumor. The WT describes the complete extent of the disease, as it entails the TC and the peritumoral edema (ED), which is typically depicted by hyper-intense signal in FLAIR. For more detailed description

Segmentation model	Description
	of tumor regions, please see the Brats2018 data page. This model was trained using a similar approach described in 3D MRI brain tumor segmentation using autoencoder regularization, which was a winning method in Multimodal Brain Tumor Segmentation Challenge (BraTS) 2018. The model was trained using BraTS 2018 training data (285 cases). This model achieves the following Dice score on the validation data (our own split from the training dataset. Check the datalist file in example folder): ▸ Tumor core (TC): 0.86 ▸ Whole tumor (WT): 0.90 ▸ Enhancing tumor (ET): 0.77 Tutorial inside of the docker (on how the model was trained): ▸ /opt/nvidia/medical/segmentation/examples/brats/tutorial_brats.ipynb Model input and output: ▸ Input: 4 channel 3D MRIs (T1c, T1, T2, FLAIR) ▸ Output: 3 channels of tumor subregion 3D masks
▸ segmentation_brain_br16__t1c2tc	The model is similar to "segmentation_brain_br16_full" model, except the input is only 1 channel MRI (T1c) and the output is only 1 channel for brain tumor subregion (TC). A pre-trained model for volumetric (3D) brain tumor segmentation (only TC from T1c images). The model was trained using BraTS 2018 training data (285 cases). https://www.med.upenn.edu/sbia/brats2018/data.html This model achieves the following Dice score on the validation data (our own split from the training dataset):

Segmentation model	Description
	‣ Tumor core (TC): 0.86 Tutorial inside of the docker (on how to use this pre-trained model and fine-tune on your own data): ‣ /opt/nvidia/medical/segmentation/examples/brats/tutorial_brats.ipynb Model input and output: ‣ Input: 1 channel MRI (T1c) ‣ Output: 1 channels 3D mask (TC)
Liver and Tumor segmentation	
‣ segmentation_liver	A pre-trained model for volumetric (3D) segmentation of the liver and lesion in portal venous phase CT image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 104 training images and 27 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ‣ Liver: 0.946 ‣ Tumor: 0.540 Model folder inside the docker: ‣ /opt/nvidia/medical/segmentation/tasks/MSD_Task03_Liver Model input and output: ‣ Input: 1 channel CT image ‣ Output: 3 channels: ‣ Label 1: liver ‣ Label 2: tumor ‣ Label 0: everything else
Hippocampus segmentation	

Segmentation model	Description
▶ segmentation_hippocampus	A pre-trained model for volumetric (3D) segmentation of the hippocampus head and body from mono-modal MRI image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 208 training images and 52 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▶ Hippocampus head: 0.858 ▶ Hippocampus body: 0.846 Model folder inside the docker: ▶ /opt/nvidia/medical/segmentation/tasks/MSD_Task04_Hippocampus Model input and output: ▶ Input: 1 channel 3D MRI image ▶ Output: 3 channels: ▶ Label 1: hippocampus head ▶ Label 2: hippocampus body ▶ Label 0: everything else
Lung Tumor segmentation	
▶ segmentation_lung	A pre-trained model for volumetric (3D) segmentation of the lung tumor from CT image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 50 training images and 13 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▶ Lung tumor: 0.383

Segmentation model	Description
	Model folder inside the docker: ▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task06_Lung Model input and output: ▸ Input: 1 channel CT image ▸ Output: 2 channels: ▸ Label 1: lung tumor ▸ Label 0: everything else
Prostrate segmentation	
▸ segmentation_prostate	A pre-trained model for volumetric (3D) segmentation of the prostate central gland and peripheral zone from the multimodal MR (T2, ADC). It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 25 training image pairs and 7 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▸ Prostate peripheral zone: 0.467 ▸ Prostate central gland: 0.519 Model folder inside the docker: ▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task05_Prostate Model input and output: ▸ Input: 2 channel MRI image ▸ Channel 1: T2 MRI ▸ Channel 2: ADC ▸ Output: 3 channels: ▸ Label 1: prostate peripheral zone ▸ Label 2: prostate central gland ▸ Label 0: everything else

Segmentation model	Description
Left atrium segmentation	
‣ segmentation_heart	A pre-trained model for volumetric (3D) segmentation of the left atrium from Mono-modal MRI. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 16 training images and 4 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ‣ Left atrium: 0.912 Model folder inside the docker: ‣ /opt/nvidia/medical/segmentation/tasks/MSD_Task02_Heart Model input and output: ‣ Input: 1 channel MRI image ‣ Output: 2 channels: ‣ Label 1: left atrium ‣ Label 0: everything else
Pancreas and tumor segmentation	
‣ segmentation_pancreas	A pre-trained model for volumetric (3D) segmentation of the pancreas and tumor from portal venous phase CT. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 224 training images and 57 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ‣ Pancreas: 0.640

Segmentation model	Description
	▸ Tumor: 0.341 Model folder inside the docker: ▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task07_Pancreas Model input and output: ▸ Input: 1 channel CT image ▸ Output: 3 channels: ▸ Label 1: pancreas ▸ Label 2: tumor ▸ Label 0: everything else
Colon tumor segmentation	
▸ segmentation_colon	A pre-trained model for volumetric (3D) segmentation of the colon tumor primaries from CT image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 100 training images and 26 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▸ Colon tumor: 0.370 Model folder inside the docker: ▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task10_Colon Model input and output: ▸ Input: 1 channel CT image ▸ Output: 2 channels: ▸ Label 1: colon tumor ▸ Label 0: everything else
Hepatic vessel and tumor segmentation	
▸ segmentation_hepatic_vessel	A pre-trained model for volumetric (3D) segmentation of the hepatic vessel

Segmentation model	Description
	and tumor from CT image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 242 training images and 61 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▸ Hepatic vessel: 0.569 ▸ Liver tumor: 0.391 Model folder inside the docker: ▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task08_HepaticVessel Model input and output: ▸ Input: 1 channel CT image ▸ Output: 3 channels: ▸ Label 1: hepatic vessel ▸ Label 2: liver tumor ▸ Label 0: everything else
Spleen segmentation	
▸ segmentation_spleen	A pre-trained model for volumetric (3D) segmentation of the spleen from CT image. It is trained using the runner-up awarded pipeline of the "Medical Segmentation Decathlon Challenge 2018" with 32 training images and 9 validation images. Please refer to the model folder inside the docker for details. And the provided training configuration required 16GB-memory GPUs. This model achieves the following Dice score on the validation data (our own split from the training dataset): ▸ Spleen: 0.957 Model folder inside the docker:

Segmentation model	Description
	▸ /opt/nvidia/medical/segmentation/tasks/MSD_Task09_Spleen Model input and output: ▸ Input: 1 channel CT image ▸ Output: 2 channels: ▸ Label 1: spleen ▸ Label 0: everything else

6.2. Classification models

Chest X-ray Classification	
classification_chestxray	A pre-trained model for disease pattern detection in chest x-rays. It is trained using PLCO training data and evaluated on the PLCO validation data. Please refer to "medical/segmentation/examples/brats/tutorial_brats.ipynb" inside the docker and the files in the same folder for details. This model achieves the following AUC score on the validation data: ▸ Averaged AUC over all disease categories: 0.8680 Model folder inside the docker: ▸ **`/opt/nvidia/medical/classification/examples/PLCO`** Model input and output: ▸ Input: 16-bit CXR png ▸ Output: 15 binary labels, each bit is corresponding to the prediction of 'Nodule', 'Mass', 'Distortion of Pulmonary Architecture', 'Pleural Based Mass', 'Granuloma', 'Fluid in Pleural Space', 'Right Hilar Abnormality', 'Left Hilar Abnormality', 'Major Atelectasis', 'Infiltrate', 'Scarring', 'Pleural Fibrosis', 'Bone/Soft Tissue Lesion', 'Cardiac Abnormality', 'COPD'

6.3. Data transforms and augmentations

Here is a list of built-in data transformation functions. If you need additional transformation functions, please contact us at the TLT user forum: http://devtalk.nvidia.com.

Transforms	Description
transforms.LoadNifty	Load NIfTI data. The value of each key (specified by *fields*) in input "dict" can be a string (a path to a single NIfTI file) or a list of strings (several paths to multiple NIfTI files, if there are several channels saved as separate files). ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. ▸ Returns: - Each field of "dict" is substituted by a 4D numpy array.
transforms.VolumeTo4dArray	Transforms the value of each key (specified by *fields*) in input "dict" from 3D to 4D numpy array by expanding one channel, if needed. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. ▸ Returns: - Each field of "dict" is substituted by a 4D numpy array.
transforms.ScaleIntensityRange	Randomly shift the intensity level of the numpy array. ▸ init_args: - field: *string* one key_value to apply, e.g. "image".

Transforms	Description
	- magnitude: *float* quantity scale of shift, has to be greater than zero.
transforms.ScaleIntensityOscillation	Randomly shift scale level for image. ▸ Args: ▸ field: key string, e.g. "image". ▸ magnitude: quantity of scale shift, has to be greater than zero. ▸ Returns ▸ Data with an offset on intensity scale. data with an offset on intensity scale.
transforms_ctrl. CropSubVolumePosNegRatioCtrlEpoch	Randomly crop the foreground and background ROIs from both the image and mask for training. The sampling ratio between positive and negative samples is adjusted with the epoch number. ▸ init_args: - image_field: *string* one key_value to apply, e.g. "image". - label_field: *string* one key_value to apply, e.g. "label". - size: *list of ints* cropped ROI size, e.g., [96, 96, 96]. - ratio_start: *float* positive/negative ratio when training start. - ratio_end: *float* positive/negative ratio when training end. - ratio_step: *float* changing of positive/negative ratio after each step. - num_epochs: *int* epochs of one step.

Transforms	Description
	▸ Returns: - Updated dictionary with cropped ROI image and mask
transforms_fastaug. TransformVolumeCropROIFastPosNegRatio	Fast 3D data augmentation method (CPU based) by combining 3D morphological transforms (rotation, elastic deformation, and scaling) and ROI cropping. The sampling ratio is specified by *pos/neg.* ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - size: *list of int* cropped ROI size, e.g., [96, 96, 96]. - deform: *boolean* whether to apply 3D deformation. - rotation: *boolean* whether to apply 3D rotation. - rotation_degree: *float* the degree of rotation, e.g., 15 means randomly rotate the image/label in a range [-15, +15]. - scale: *boolean* whether to apply 3D scaling. - scale_factor: *float* the percentage of scaling, e.g., 0.1 means randomly scaling the image/ label in a range [-0.1, +0.1]. - pos: *float* the factor controlling the ratio of positive ROI sampling. - neg: *float* the factor controlling the ratio of negative ROI sampling. ▸ Returns:

Transforms	Description
	- Updated dictionary with cropped ROI image and mask after data augmentation.
transforms.AdjustContrast	Randomly adjust the contrast of the *field* in input "dict". ▸ init_args: - field: *string* one key_value to apply, e.g. "image".
transforms.AddGaussianNoise	Randomly add Gaussian noise to the *field* in input "dict". ▸ init_args: - field: *string* one key_value to apply, e.g. "image".
transforms.LoadPng	Load png image and the label. The value of "image" must be a string (a path to a single png file) while the value of the "label" must be a list of labels. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. ▸ Returns: - "image" of "dict" is substituted by a 3D numpy array while the "label" of "dict" is substituted by a numpy list
transforms.CropRandomSubImageInRange	Randomly crop 2D image. The crop size is randomly selected between lower_size and image size. ▸ init_args: - lower_size: *int or float* lower limit of crop size, if float, then must be fraction <1 - max_displacement: *float* max displacement from center to crop

Transforms	Description
	- keep_aspect: *boolean* if true, then original aspect ratio is kept ▸ Returns: - The "image" field of input "dict" is substituted by cropped ROI image.
transforms.NPResizeImage	Resize the 2D numpy array (channel x rows x height) as an image. ▸ init_args: - applied_keys: *string* one key_value to apply, e.g. "image". - output_shape: *list of int with length 2* e.g., [256,256]. - data_format: *string* ''channels_first', 'channels_last', or 'grayscale'.
transforms.NP2DRotate	Rotate 2D numpy array, or channelled 2D array. If *random* is set to true, then rotate within the range of [*-angle, angle*] ▸ init_args: - applied_keys: *string* one key_value to apply, e.g. "image". - angle: *float* e.g. 7. - random: *boolean* default is false.
transforms.NPExpandDims	Add a singleton dimension to the selected axis of the numpy array. ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - expand_axis: *int*

Transforms	Description
	axis to expand, default is 0
transforms.NPRepChannels	Repeat a numpy array along specified axis, e.g., turn a grayscale image into a 3-channel image. ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - channel_axis: *int* the axis along which to repeat values. - repeat: *int* the number of repetitions for each element.
transforms.CenterData	Center numpy array's value by subtracting a subtrahend and dividing by a divisor. ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - subtrahend: *float* subtrahend. If None, it is computed as the mean of dict[key_value]. - divisor: *float* divisor. If None, it is computed as the std of dict[key_value]
transforms.NPRandomFlip3D	Flip the 3D numpy array along random axes with the provided probability. ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - probability: *float*

Transforms	Description
	probability to apply the flip, value between 0 and 1.0.
transforms.NPRandomZoom3D	Apply a random zooming to the 3D numpy array. ▸ init_args: - applied_keys: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - lower_limits: *list of float* lower limit of the zoom along each dimension. - upper_limits: *list of float* upper limit of the zoom along each dimension. - data_format: *string* 'channels_first' or "channels_last". - use_gpu: *boolean* whether to use cupy for GPU acceleration. Default is false. - keep_size: *boolean* default is false which means this function will change the size of the data array after zooming. Setting keep_size to True will result in an output of the same size as the input.
transforms.CropForegroundObject	Crop the 4D numpy array and resize. The numpy array must have foreground voxels. ▸ init_args: - size: *list of int* resized size. - image_field: *string* "image". - label_field: *string*

Transforms	Description
	"label".
transforms.NPRandomRot90_XY	Rotate the 4D numpy array along random axes on XY plane (axis = (1, 2)). ▸ init_args: - applied_keys: *string* one key_value to apply, e.g. "image". - probability: *float* probability to utilize the transform, between 0 and 1.0.
transforms.AddExtremePointsChannel	Add and additional channel to 4D numpy array where the extreme points of the foreground labels are modeled as Gaussians. ▸ init_args: - image_field: *string* "image". - label_field: *string* "label". - sigma: *float* size of Gaussian. - pert: *boolean* random perturbation added to the extreme points.
transforms.NormalizeNonzeroIntensities	Normalize 4D numpy array to zero mean and unit std, based on non-zero elements only for each input channel individually. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"].
transforms.SplitAcrossChannels	Splits the 4D numpy array across channels to create new dict entries. New key_values shall be *applied_key*+channel number, e.g. "image1".

Transforms	Description
	▸ init_args: - applied_key: *string* one key_value to apply, e.g. "image".
transforms.LoadResolutionFromNifty	Get the image resolution from an NifTI image ▸ init_args: - applied_key: *string* one key_value to apply, e.g. "image". ▸ Returns: - "dict" has a new key-value pair: dict[applied_key+"_resolution"]: resolution of the NIfTI image
transforms.Load3DShapeFromNumpy	Get the image shape from an NifTI image ▸ init_args: - applied_key: *string* one key_value to apply, e.g. "image". ▸ Returns: - "dict" has a new key-value pair: dict[applied_key+"_shape"]: shape of the NIfTI image
transforms.ResampleVolume	Resample the 4D numpy array from current resolution to a specific resolution ▸ init_args: - applied_key: *string* one key_value to apply, e.g. "image". - resolution: *list of float* input image resolution. - target_resolution: *list of float* target resolution.
transforms.WriteNiftyResults	Save 3D numpy array to a NifTI file. ▸ init_args: - applied_key: *string*

Transforms	Description
	one key_value to apply, e.g. "image". - write_path: *string* path to store the output image. ▸ Returns: - Write the numpy array to <write_path>/<input_file_name>/<input_file_name>_<applied_key>.nii
transforms.BratsConvertLabels	Brats data specific. Convert input labels format (indices 1,2,4) into proper format. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"].
transforms. CropSubVolumeRandomWithinBounds	Crops a random subvolume from within the bounds of 4D numpy array. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - size: *list of int* the size of the crop region e.g. [224,224,128].
transforms.FlipAxisRandom	Flip the numpy array along its dimensions randomly. ▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - axis : *list of ints* which axes to attempt to flip (e.g. [0,1,2] - for all 3 dimensions) - the axis indices must be provided only for spatial dimensions.
transforms.CropSubVolumeCenter	Crops a center subvolume from within the bounds of 4D numpy array.

Transforms	Description
	▸ init_args: - fields: *string or list of strings* key_values to apply, e.g. ["image", "label"]. - size: *list of int* the size of the crop region e.g. [224,224,128] (similar to CropSubVolumeRandomWithinBounds, but crops the center)