NVIDIA CLARA TRAIN SDK: AI-ASSISTED ANNOTATION

DU-09358-002 _v2.0 | March 2019

Getting Started Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

The NVIDIA Clara Train SDK with AI-Assisted Annotation uses deep learning techniques to take points of interest drawn by radiologists to approximate points as input along with the 3D volume data to return an auto-annotated set of slices. Auto-annotation step is achieved using NVIDIA's pre-trained deep learning models for different organs. Neither application developers nor radiologists need to have knowledge of deep learning knowledge to benefit from NVIDIA's deep learning expertise out of the box.

The Clara Train SDK works in conjunction with Transfer Learning Toolkit for medical imaging. If you are a developer or an engineer developing medical image analysis applications for healthcare providers, this guide can help you get started with integrating AI-Assisted Annotation SDK into your existing custom applications or into existing medical imaging applications such as MITK, or ITK-Snap, without any prior deep learning knowledge.

Chapter 2. REQUIREMENTS AND INSTALLATION OF THE SDK

Using the Clara Train SDK 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.

Installation Prerequisites

- NVIDIA GPU driver v410.xx or above. Download from https://www.nvidia.com/Download/index.aspx?lang=en-us.
- Install the Nvidia Docker 2.0 from: https://github.com/NVIDIA/nvidia-docker.

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`**

Chapter 3.
ACCESSING PRE-TRAINED MODELS

3.1. Downloading the models

Use these steps to download the pre-trained models.

Please use the API key that was generated when downloading the container. If you did not get an API key yet, follow the instructions in Requirements and installation of the SDK.

Optionally, You can export **`API_KEY=<your_key>`** and use the **`$API_KEY`** environment variable.

tlt-pull

`tlt-pull` is a tool included in the container. Therefore, you have to be inside a running container to access tlt-pull. The easiest way is described in Fine-tuning from a pre-trained model, which shows examples on how to use tlt-pull from the Jupyter Notebook. If you are familiar with Docker container, you can launch the container with this command to get into the container and access tlt-pull directly.

```
docker run -v /host/path/to/save/models:/var/tmp/models --
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 -v /host/path/to/save/models:/var/tmp/models --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/nvidia/
tlt-annotate:v0.1-py3 /bin/bash
```

Getting a list of models

Use this command to get a list of models that are available.

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

All the models that start with the annotation_ keyword can be used with the annotation server.

Getting a list of model versions

Here's an example of using this command to get a list of model versions for annotation_spleen.

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

Downloading a model

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

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

Example output from using this command.

```

tlt-pull -k $API_KEY -m annotation_spleen -v 1 -d /var/tmp/models/
annotation_spleen -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 annotation_spleen.
Downloaded files are in /var/tmp/models/annotation_spleen/.
```

See Available annotation models for a list of available models

3.2. Convert models to TRT optimized models

Once the necessary models are downloaded, convert them to TRT optimized model by using this command (assuming the downloaded checkpoint is saved to the folder specified by CHECKPOINT):

```
CHECKPOINT=/var/tmp/models/annotation_spleen
```

```
tlt-export --model_name 'model' --model_file_path ${CHECKPOINT} --
input_node_names 'cond/Merge' --output_node_names 'NV_OUTPUT_LABEL'
```

For more information see, Exporting the model to a TensorRT optimized model inference.

Chapter 4. USING THE ANNOTATION SERVER

This chapter describes tasks and information needed to use the Annotation server.

4.1. Starting the annotation server

The models used in the annotation server need to be TRT-optimized models. For an example of how to convert models downloaded from NGC see, Convert models to TRT optimized models .

`engine_pool.json` is one of the configuration file the annotation server requires. Typically, this is created on the host machine where the docker is started.

```
[
    {
        "name": "MyAnnotationSpleenModel",
        "class": "Dextr3dCroppedEngine",
        "module": "medical.aas.dextr3dcropped",
        "config_file": "/var/tmp/models/annotation_spleen.json",
        "labels": ["spleen"]
    },
    {
        "name": "MyAnnotationLiverModel",
        "class": "Dextr3dCroppedEngine",
        "module": "medical.aas.dextr3dcropped",
        "config_file": "/var/tmp/models/annotation_liver.json",
        "labels": ["liver"]
    }
]
```

The json file includes a list of json objects. In this case there are two objects included. The combination of class and module will identify which class is used to construct the model, as well as the pre/post processings used. The **`config_file`** will be loaded by the class for its internal consumption, such as parameters and other related information. The labels field value is a list of label names used to train the model. The name field value must be unique.

4.1.1. Creating configuration files for the annotation models

The annotation server loads all models specified by the **`engine_pool.json`** file with each model having its own configuration file.

The model configuration JSON file for this example is assumed to be /var/tmp/models/annotation_spleen.json. Here is the example model configuration JSON.

The configuration file is shown as an example. For a list of available pre-trained annotation models and their configuration files see, Available annotation models.

```
{
    "model_format": "trtmodel",
    "model_path": "/var/tmp/models/annotation_spleen/model.trt.pb",
    "thresh": 0.5,
    "result_dir": "/var/tmp/experiments/aiaa",
    "input_nodes":
    {
        "image": "cond/Merge"
    },
    "output_nodes":
    {
        "model": "NV_OUTPUT_LABEL"
    },
    "batch_size": 1,
    "pre_transforms":
    [
        {
            "name": "transforms.VolumeTo4dArray",
            "fields": ["image", "label"]
        },
        {
            "name": "transforms.NormalizeNonzeroIntensities",
            "fields": "image"
        },
        {
            "name": "transforms.AddExtremePointsChannel",
            "image_field": "image",
            "label_field": "label",
            "sigma": 3,
            "pert": 0
        }
    ],
    "post_transforms":
    [
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "model",
            "channel_names": ["background", "prediction"]
        }
    ],
    "writers":
    [
        {
            "applied_key": "prediction",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        }
    ]
}
```

If you are not inside the container already, start the container using the docker run command:

```
docker run --runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=0 -it --rm -p 5000:5000
 -v /var/tmp/models:/var/tmp/models nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3 /
bin/bash
```

When you are in the docker terminal, you can start the annotation server by running the following command.

```
/opt/nvidia/medical/aas/start_aas.sh --config /var/tmp/models/engine_pool.json
```

If you would like to train your on model, see the End-to-end tutorial.

4.2. Confirm that the annotation server is working

The annotation server responds to the models API to list all models currently loaded. Execute this command:

```
curl http://$MACHINE_IP:5000/v1/models
```

Please replace $MACHINE_IP with actual IP address of your machine.

This an examples of the results:

```
[{"labels": ["brain_tumor_core"], "internal name": "Dextr3dCroppedEngine",
 "decription": "", "name": "Dextr3DBrainTC"}, {"labels": ["liver"], "internal
 name": "Dextr3dCroppedEngine", "decription": "", "name": "Dextr3DLiver"},
 {"labels": ["brain_whole_tumor"], "internal name": "Dextr3dCroppedEngine",
 "decription": "", "name": "Dextr3DBrainWT"}]
```

4.3. Integrating into your own application

Once the annotation server is setup and started with specific organ models, client component of the SDK is delivered as an open source reference implementation to demonstrate the integration process. Client code and libraries are provided for both C++ and Python languages on NVIDIA Github page here: https://github.com/NVIDIA/ai-assisted-annotation-client

Chapter 5. PREPARING DATA FOR SEGMENTATION

In case you want to train and deploy a new model using your own data, you will need to use the Transfer Learning Toolkit.

This section describes the format in which the data can be used for the TLT toolkit for 3D segmentation. For more information see the End-to-end tutorial.

5.1. Data format

All input images and labels must be in NIfTI format at isotropic resolution (e.g. 1x1x1mm). 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, for example DICOM, or if you want to convert the image and label mask to isotropic resolution (e.g. to finetune with the pre-trained models), you can use the provided Data Converter or any other software of your own choice, such as ITK-SNAP or directly in python, to convert the data to NIfTI isotropic resolution (the same resolution in all three dimensions of the image).

5.2. Datalist JSON file

The JSON file that describes 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 denoting a path to a single NIfTI file with dense segmentation mask/masks. Only mono-channel images are currently supported.

The value for the **`label`** key, must be a string (a path to a single NIfTI file with dense segmentation mask/masks). The label file can define segmentation using indices. Multichannel one-hot-encoded images are not supported for AIAA training.

The **`validation`** key is optional, and if provided the corresponding images/labels will be used to compute the validation metrics at the end of each training epoch. The validation metrics are computed with varying frequencyif specified in the main training config. The **`validation`** section does not need to include the label keys, if the **`datalist.json`** file is used to compute the output segmentation masks.

5.3. Data structure

The layout of data files can be arbitrary. The JSON file describing the data list should contain relative paths to all data files. It can be placed in the top directory of the data folder, or the data folder root path can be specified during training or inference. For example for a data structure below.

```
|--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
```

The **`datalist.json`** file will be similar to this. All the paths are relative to the **`datalist.json`** file 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.

5.4. 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.

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.

Chapter 6. TRAINING THE MODEL

This chapter discusses using the **`tlt-train`** command in the Transfer Learning Toolkit to train models with single and multiple GPUs.

6.1. Training from scratch or fine-tuning an annotation model

Segmentation training

Use the **`tlt-train segmentation`** command to perform AIAA training or to fine-tune 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.
$DATA_LIST	Path to a JSON file with a list of input data files.
$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 the previously pretrained weights. If it is not provided, the model is initialized from scratch with random weights initialization. The checkpoint file can your own previously trained checkpoint or one of the provided pretrained 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

Here's an example of a single GPU training session, the command line would look like this.

$TRAIN_CONFIG - is main configuration file. Here's an AIAA example for training an annotation on CT images:

```
{
    "num_channels": 2,
    "num_classes": 2,
    "batch_size": 2,
    "epochs": 2000,
    "num_workers": 8,
    "multi_gpu":false,
    "network_input_size": [
      128,
      128,
      128
    ],
    "num_training_epoch_per_valid": 10,
    "use_scanning_window": true,
  "net_config":
    {
        "name": "SegmAhnet3D",
        "if_from_scratch": false,
        "if_use_psp": false,
        "pretrain_weight_name": "/var/tmp/
resnet50_weights_tf_dim_ordering_tf_kernels.h5",
        "plane": "z",
        "final_activation": "softmax"
    },
  "train" : {
    "loss": {
      "name": "losses.dice_loss"
    },
    "optimizer": {
      "name": "Adam",
      "lr": 1e-3,
      "epsilon": 1e-4
    },
    "lr_policy": {
      "name": "lr_policy.DecayLRonStep",
      "decay_ratio": 0.1,
      "decay_freq": 50000
    },
    "pre_transforms":
    [
      {
          "name": "transforms.LoadNumpyFromFilename",
          "fields": ["image", "label"]
      },
      {
          "name": "transforms.NPRandomFlip3D",
          "applied_keys": ["image", "label"]
      },
      {
          "name": "transforms.NPRandomZoom3D",
          "applied_keys": ["image", "label"],
          "lower_limits": [0.8, 0.8, 0.8],
          "upper_limits": [1.2, 1.2, 1.2]
      },
      {
          "name": "transforms.VolumeTo4dArray",
          "fields": ["image", "label"]
      },
      {
          "name": "transforms.ScaleIntensityRange",
          "field": "image",
          "a_min": -1024,
          "a_max": 1024,
          "b_min": -1.0,
          "b_max": 1.0,
          "clip": true
      },
      {
          "name": "transforms.ScaleIntensityOscillation",
          "field": "image",
          "magnitude": 0.2
      },
      {
          "name": "transforms.CropForegroundObject",
          "size": [128, 128, 128],
          "image_field": "image",
          "label_field": "label",
          "pad": 20,
          "foreground_only": true
      },
      {
          "name": "transforms.NPRandomRot90_XY",
          "applied_keys": ["image", "label"]
      },
      {
          "name": "transforms.AddExtremePointsChannel",
          "image_field": "image",
          "label_field": "label",
          "sigma": 3,
          "pert": 3
      }
    ]
  },
  "validate" : {
    "metrics": [
      {
        "name": "MetricAverageFromArrayDice",
        "tag": "mean_dice",
        "stopping_metric": true,
        "applied_key": "model"
      }
    ],
    "pre_transforms": [
      {
          "name": "transforms.LoadNumpyFromFilename",
          "fields": ["image", "label"]
      },
      {
          "name": "transforms.VolumeTo4dArray",
          "fields": ["image", "label"]
      },
      {
          "name": "transforms.ScaleIntensityRange",
          "field": "image",
          "a_min": -1024,
          "a_max": 1024,
          "b_min": -1.0,
          "b_max": 1.0,
          "clip": true
      },
      {
          "name": "transforms.CropForegroundObject",
          "size": [128, 128, 128],
          "image_field": "image",
          "label_field": "label",
          "pad": 20,
          "foreground_only": true
      },
      {
          "name": "transforms.AddExtremePointsChannel",
          "image_field": "image",
          "label_field": "label",
          "sigma": 3,
          "pert": 0
      }
    ]
  }
}
```

6.2. Multi-GPU training

Use **`tlt-train segmentation`** to run the training using multi-GPU:

```
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 |& 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`** should be set to true for example, **`"multi_gpu":true`**.

6.3. Tensorboard visualization

You can use this command to use Tensorboard for visualization:

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

6.4. End-to-end tutorial

6.4.1. Running the demo in the notebook

Run this demo using the provided tutorial_spleen.ipynb notebook.

First run the docker.

```
docker run --runtime=nvidia -it --rm -v /your_local_path/aiaa_demo:/mnt/
aiaa_spleen -p 8888:8888 $DOCKER_IMAGE jupyter notebook  /opt/nvidia/medical/
annotation/examples/MSD_Task09_Spleen --ip 0.0.0.0 --allow-root --no-browser
```

Copy the URL together with the token as shown on the terminal into a web-browser once the docker run command above is executed.

Open the **notebook tutorial_spleen.ipynb** in your browser. Here **/your_local_path/aiaa_demo** is your own local folder to store data, pre-trained models, and output files. **$DOCKER_IMAGE** is a docker image name such as **nvcr.io/nvtltea/med/tlt-annotate:v0.1-py3**.

Alternatively, follow the instructions below to reproduce steps in the notebook manually.

6.4.2. Training on an annotation model from scratch or fine-tune

- Input: 3D CT images, 3D label images
- Output: trained annotation model

6.4.2.1. Start the docker in interactive mode

Start the docker in console/interactive mode.

```
docker run --runtime=nvidia --network=host -it --rm -v /your_local_path/
aiaa_demo:/mnt/aiaa_spleen -p 8888:8888 -w /opt/nvidia/medical/annotation/
examples/MSD_Task09_Spleen $DOCKER_IMAGE /bin/bash
```

6.4.2.2. Define paths to config files and prepare data

You can define paths to configuration files and data. It's also possible to supply these paths directly to the scripts, or you can define them as variables.

```
# data root
FILE_ROOT=/mnt/aiaa_spleen
# json config with a list of data files for training and validation (MSD data in
 this case)
DATA_LIST=/opt/nvidia/medical/annotation/examples/MSD_Task09_Spleen/
Task09_Spleen_dataset.json
# output directory to save outputs (such as Tensorboard logs and saved graph/
checkpoints)
TRAIN_OUTPUT_DIR=/mnt/aiaa_spleen/output
# json config with training configuration
TRAIN_CONFIG=/opt/nvidia/medical/annotation/examples/MSD_Task09_Spleen/
config_train_aiaa_Task09_Spleen.json
```

Data

The Medical Segmentation Decathlon (MSD) challenge Task09_Spleen dataset is used for the tutorial See http://medicaldecathlon.com/ for more information.

Download the data from the MSD website (https://drive.google.com/file/d/1jzeNU1EKnK81PyTsrx0ujfNl-t0Jo8uE/view?usp=sharing) and place in the directory mounted at $FILE_ROOT.

For other datasets, the relative file paths in $DATA_LIST would have to be updated accordingly.

Extract the image files (and remove unnecessary files, ignore the warnings about removing directories):

```
tar -xvf ${FILE_ROOT}/Task09_Spleen.tar --directory ${FILE_ROOT} && rm
 ${FILE_ROOT}/Task09_Spleen/*/.*
```

Data preparation

All MSD challenge datasets are in NIfTI format, but need to be resampled to isotrophic 1x1x1 mm pixel spacing. Data conversion is needed to resample the images and labels.

You are converting both 3D volumetric images and their segmentation labels, so run the converter utility **`tlt-dataconverttwice`** for images and for labels with **`-l`** flag).

```
# Resample images (this will take some time)
tlt-dataconvert -d ${FILE_ROOT}/Task09_Spleen/imagesTr -r 1.0 -s .nii.gz -
e .nii.gz -o ${FILE_ROOT}/Task09_Spleen_res1.0/imagesTr
```

```
# Resample labels (notice the `-l` argument; this will also take some time...)
tlt-dataconvert -l -d ${FILE_ROOT}/Task09_Spleen/labelsTr -r 1.0 -s .nii.gz -
e .nii.gz -o ${FILE_ROOT}/Task09_Spleen_res1.0/labelsTr
```

The JSON file $DATA_LIST lists the resampled training and validation split used in this example. The data JSON file looks like the following.

The $DATA_LIST contains relative paths with respect to the root directory of the data, that is $FILE_ROOT.

```
cat $DATA_LIST
```

6.4.2.3. Run training

Configuration

The json config file **`$TRAIN_CONFIG`** specifies the training options, including data I/O transforms, data augmentation transforms, model, loss, optimizer and hyper parameters. The training config file looks like:

```
cat $TRAIN_CONFIG
```

With data index file $DATA_LIST and $TRAIN_CONFIG, the tlt-train command can be used to run model training and saves the outputs, including tensorboard log and checkpoints, to the $OUTPUT_DIR.

There are two options for training:

- Training from scratch
- Fine-tuning from a pre-trained model

6.4.2.3.1. Training from scratch

The next command illustrates how to train a model from scratch. If you want to fine-tune from an existing model instead, continue to fine-tuning.

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

6.4.2.3.2. Fine-tuning from a pre-trained model

If you have already trained a model from scratch continue to Convert the checkpoint to TensorRT optimized model.

If you want to use one of the pre-trained models for fine-tuning, you can use the **`tlt-pull`** command to download the model. You will have to specify your NGC API key in **`YOUR_NGC_API_KEY`**.

For fine-tuning, NVIDIA recommends reducing the initial learning rate in **`TRAIN_CONFIG`** to**`"lr": 1e-6`** for example. In this case, use **`TRAIN_FINETUNE_CONFIG`** where the learning rate has been already adjusted.

```
# Choose the model to download
MODEL_NAME=annotation_spleen
VERSION=1
API_KEY=YOUR_NGC_API_KEY
TRAIN_FINETUNE_CONFIG=/opt/nvidia/medical/annotation/examples/MSD_Task09_Spleen/
config_train_finetune_aiaa_Task09_Spleen.json
PRETRAINED_CHECKPOINT_DIR=/mnt/aiaa_spleen/pretrained_model
```

```
# download the model
tlt-pull -m $MODEL_NAME -v $VERSION -k $API_KEY -d $PRETRAINED_CHECKPOINT_DIR -o
 nvtltea -t med;
```

When downloading the model is complete start fine-tuning the model to the current dataset.

```
tlt-train segmentation \
    --config $TRAIN_CONFIG \
    --file_root $FILE_ROOT \
    --data_list $DATA_LIST \
    --checkpoint $PRETRAINED_CHECKPOINT_DIR/model.ckpt \
    --model_dir $TRAIN_OUTPUT_DIR
```

6.4.2.4. Convert the checkpoint to TensorRT optimized model

Once a model is trained, use the **`tlt-export`** command to export the model for inference or evaluation.

```
# checkpoint dir is the same as TRAIN_OUTPUT_DIR where we saved the checkpoint
env CHECKPOINT=/mnt/aiaa_spleen/output

# export the optimized model
tlt-export \
    --model_name 'model' \
    --model_file_path ${CHECKPOINT} \
    --input_node_names 'cond/Merge' \
    --output_node_names 'NV_OUTPUT_LABEL'
```

6.4.2.5. Predict using the trained model

When training is complete, use the **`tlt-infer`** command to run inference using the trained model. Inference requires its own config file, corresponding to the training configuration file. Here's an example:

```
# Inference variables
env INFERENCE_CONFIG=/mnt/aiaa_spleen/config_infer_aiaa_Task09_Spleen.json
# checkpoint dir is the same as TRAIN_OUTPUT_DIR where we saved the checkpoint
env CHECKPOINT=/mnt/aiaa_spleen/output
env INFERENCE_OUTPUT_PATH=/mnt/aiaa_spleen/output/inference
```

```
# show inference configuration
cat $INFERENCE_CONFIG
```

Now run inference. Inference runs the trained checkpoint model, on files in your DATA_LIST validation section. You can assume that images with ground truth labels here, so that you can simulate user-clicks for the AIAA model.

```
echo "Run inference on "$DATA_LIST
mkdir $INFERENCE_OUTPUT_PATH
tlt-infer \
    --model_save_path $CHECKPOINT/model.trt.pb \
    --config $INFERENCE_CONFIG \
    --model_format "trtmodel" \
    --file_root $DATA_ROOT \
    --output_path $INFERENCE_OUTPUT_PATH \
    --data_file $DATA_LIST \
    --setup standard
```

6.4.2.6. Evaluate the trained model

When the training is complete, use the **`tlt-evaluate`** command to generate an evaluattion report.

```
# Evaluation variables
EVAL_CONFIG=/opt/nvidia/medical/annotation/examples/MSD_Task09_Spleen/
config_eval_aiaa_Task09_Spleen.json
# checkpoint dir is the same as TRAIN_OUTPUT_DIR where we saved the checkpoint
CHECKPOINT=/mnt/aiaa_spleen/output
EVAL_OUTPUT_PATH=/mnt/aiaa_spleen/output/eval
EVAL_REPORT=/mnt/aiaa_spleen/output/eval/mean_dice_raw_results.txt
SUMMARY_REPORT=/mnt/aiaa_spleen/output/eval/mean_dice_summary_results.txt
```

```
# show evaluation configuration
cat $EVAL_CONFIG
```

Evaluation runs the trained checkpoint model, on files in your DATA_LIST validation section and compute the average Dice score per file per class. The ground truth segmentation must be available.

```
echo "Run evaluation on "$DATA_LIST
mkdir $EVAL_OUTPUT_PATH
tlt-evaluate \
    --model_save_path $CHECKPOINT/model.trt.pb \
    --config $EVAL_CONFIG \
    --model_format "trtmodel" \
    --file_root $FILE_ROOT \
    --output_path $EVAL_OUTPUT_PATH \
    --data_file $DATA_LIST \
    --setup standard
```

Use this command generates an evaluation and summary report at **`$EVALUATION_REPORT & $SUMMARY_REPORT`**. Here's an example:

```
cat $EVALUATION_REPORT
cat $SUMMARY_REPORT
```

You can run this command to use Tensorboard for visualization of the training progress.

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

Chapter 7. 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 for either validation or inference.

This command accepts the Checkpoint format of the model file as input, and produces a regular frozen graph and a TRT-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. Two formats are supported: 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`**.

Options	Description
`--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
  ```

Chapter 8. RUNNING INFERENCE

The **`tlt-infer`** tool produces a prediction of segmentation on a list of images.

8.1. Model Inference

You can run inference after exporting a training checkpoint to a TRT-optimized model (model.trt.pb).

```
# Evaluation and inference DIR variables
INFERENCE_CONFIG=./config_infer_aiaa_Task09_Spleen.json
CHECKPOINT=./output/model.trt.pb
INFERENCE_OUTPUT_PATH=./inference
EVALUATION_REPORT=./mean_dice_raw_results.txt
echo "Run inference on "$DATA_LIST
mkdir $INFERENCE_OUTPUT_PATH
tlt-infer \
    --model_save_path $CHECKPOINT/model.trt.pb \
    --config $INFERENCE_CONFIG \
    --model_format "trtmodel" \
    --file_root $DATA_ROOT \
    --output_path $INFERENCE_OUTPUT_PATH \
    --data_file $DATA_LIST \
    --setup standard
```

The **`config_infer_aiaa_Task09_Spleen.json`** an example of a configuration file used for inference only:

```
{
    "input_nodes":
    {
        "image": "cond/Merge"
    },
    "output_nodes":
    {
        "model": "NV_OUTPUT_LABEL"
    },
    "batch_size": 1,
    "pre_transforms":
    [
        {
            "name": "transforms.LoadNifty",
            "fields": ["image", "label"]
        },
        {
            "name": "transforms.CropForegroundObject",
            "size": [128, 128, 128],
            "image_field": "image",
            "label_field": "label",
            "pad": 20,
            "foreground_only": true
        },
        {
            "name": "transforms.ScaleIntensityRange",
            "field": "image",
            "a_min": -1024,
            "a_max": 1024,
            "b_min": -1.0,
            "b_max": 1.0,
            "clip": true
        },
        {
            "name": "transforms.AddExtremePointsChannel",
            "image_field": "image",
            "label_field": "label",
            "sigma": 3,
            "pert": 0
        }
    ],
    "post_transforms":
    [
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "model",
            "squeeze": true,
            "channel_names": ["background", "prediction"]
        },
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "image",
            "squeeze": true,
            "channel_names": ["image", "extreme_points"]
        }
    ],
    "writers":
    [
        {
            "applied_key": "prediction",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        },
        {
            "applied_key": "label",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "image",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        },
        {
            "applied_key": "extreme_points",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        }
    ]
}
```

Chapter 9. EVALUATING THE MODEL

This section describes how to use **`tlt-evaluate`** to run an evaluation of a list of images based on a set of ground truth labels.

9.1. Model evaluation with ground truth

The steps to evaluate with the AIAA model are similar to inference only, but with some added configurations.

```
# Evaluation and inference DIR variables
EVAL_CONFIG=./config_eval_aiaa_Task09_Spleen.json
CHECKPOINT=./output/model.fzn.pb
EVAL_OUTPUT_PATH=./eval
EVAL_REPORT=./eval/mean_dice_raw_results.txt
SUMMARY_REPORT=./eval/mean_dice_summary_results.txt
echo "Run evaluate on "$DATA_LIST
mkdir $INFERENCE_OUTPUT_PATH
tlt-evaluate \
    --model_save_path $CHECKPOINT/model.fzn.pb \
    --config $EVAL_CONFIG \
    --model_format "trtmodel" \
    --file_root $DATA_ROOT \
    --output_path $EVAL_OUTPUT_PATH \
    --data_file $DATA_LIST \
    --setup standard
```

This command generates an evaluation & summary report at $EVALUATION_REPORT & $SUMMARY_REPORT.

```
cat $EVAL_REPORT
cat $SUMMARY_REPORT
```

The configure file for inference and evaluation would look like this:

config_eval_aiaa_Task09_Spleen.json

```
{
    "input_nodes":
    {
        "image": "cond/Merge"
    },
    "output_nodes":
```

```
    {
        "model": "NV_OUTPUT_LABEL"
    },
    "batch_size": 1,
    "pre_transforms":
    [
        {
            "name": "transforms.LoadNifty",
            "fields": ["image", "label"]
        },
        {
            "name": "transforms.CropForegroundObject",
            "size": [128, 128, 128],
            "image_field": "image",
            "label_field": "label",
            "pad": 20,
            "foreground_only": true
        },
        {
            "name": "transforms.ScaleIntensityRange",
            "field": "image",
            "a_min": -1024,
            "a_max": 1024,
            "b_min": -1.0,
            "b_max": 1.0,
            "clip": true
        },
        {
            "name": "transforms.AddExtremePointsChannel",
            "image_field": "image",
            "label_field": "label",
            "sigma": 3,
            "pert": 0
        }
    ],
    "post_transforms":
    [
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "model",
            "squeeze": true,
            "channel_names": ["background", "prediction"]
        },
        {
            "name": "transforms.SplitAcrossChannels",
            "applied_key": "image",
            "squeeze": true,
            "channel_names": ["image", "extreme_points"]
        }
    ],
    "writers":
    [
        {
            "applied_key": "prediction",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        },
        {
            "applied_key": "label",
            "name": "writers.WriteNiftyResults",
            "dtype": "uint8"
        },
        {
            "applied_key": "image",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        },
        {
            "applied_key": "extreme_points",
            "name": "writers.WriteNiftyResults",
            "dtype": "float32"
        }
    ],
    "val_metrics":
    [
        {
            "name": "MetricAverageFromArrayDice",
            "tag" : "mean_dice",
            "applied_key": "prediction",
            "label_key": "label"
        }
    ]
}
```

Chapter 10. APPENDIX

10.1. Available annotation models

Here are the list of pre-trained AIAA annotation models available on NGC. Configuration files for using the models in the annotation server are inside the docker in **`/opt/nvidia/medical/aas/configs`**.

Annotation model	Description
Brain tumor annotation	
▸ annotation_brain_t1ce_tc	A pre-trained model for volumetric (3D) annotation 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 the "tumor core" (TC) based on T1c (T1 contrast en. The ET is described by areas that show hyper-intensity in T1Gd when compared to T1, but also when compared to "healthy" white matter in T1Gd. 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

Annotation 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 annotation using autoencoder regularization", which was a winning method in Multimodal Brain Tumor annotation Challenge (BraTS) 2018. The model was trained using BraTS 2018 training data (285 cases). This model achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Tumor core (TC): 0.848 Model folder inside of the docker. ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/BraTS/BraTS_tumor_core Model input and output: ▸ Input: 1 channel 3D MRIs (T1c) ▸ Output: 2 channels for background & foreground
▸ annotation_brain_t2_wt	The model is similar to "annotation_brain_t1ce_tc" model, except the input in another MRI channel is used as input (T2) and the output is the "whole tumor" region (WT). The model was trained using BraTS 2018 training data (285 cases). https://www.med.upenn.edu/sbia/brats2018/data.html This model achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Tumor core (WT): 0.884 Model folder inside the docker. ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/BraTS/BraTS_whole_tumor Model input and output:

Annotation model	Description
	▸ Input: 1 channel 3D MRI (T2) ▸ Output: 2 channels for background & foreground
Liver annotation	
▸ annotation_liver	A pre-trained model for volumetric (3D) annotation of the liver in portal venous phase CT image. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Liver: 0.958 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/ dextr3D/tasks/MSD_Task03_Liver Model input and output: ▸ Input: 1 channel CT image ▸ Output: 2 channels for background & foreground
Liver tumor annotation	
▸ annotation_livertumor	A pre-trained model for volumetric (3D) annotation of liver tumors in portal venous phase CT image, combining to MSD datasets (Task03_Liver and Task08_HepaticVessel). It is trained using our AIAA 3D model (dextr3D) with 346(104+242) training images and 88(27+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):

Annotation model	Description
	▶ Liver: 0.697 Model folder inside the docker: ▶ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task03-08_LiverTumor Model input and output: ▶ Input: 1 channel CT image ▶ Output: 2 channels for background & foreground
Hippocampus annotation	
▶ annotation_hippocampus	A pre-trained model for volumetric (3D) annotation of the hippocampus (head & body) from mono-modal MRI image. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▶ Hippocampus (body & head): 0.886 Model folder inside the docker: ▶ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task04_Hippocampus Model input and output: ▶ Input: 1 channel 3D MRI image ▶ Output: 2 channels for background & foreground
Lung tumor annotation	
▶ annotation_lung	A pre-trained model for volumetric (3D) annotation of the lung tumor from CT image. It is trained using our AIAA 3D model (dextr3D) with 50 training images and 13 validation images. Please refer to the model folder inside the docker

Annotation model	Description
	for details. And the provided training configuration required 16GB-memory GPUs. This model achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Lung tumor: 0.797 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task06_Lung Model input and output: ▸ Input: 1 channel CT image ▸ Output: 2 channels for background & foreground
Prostate annotation	
▸ annotation_prostate_cg	A pre-trained model for volumetric (3D) annotation of the prostate central gland from the T2 MRI (we ignore the ADC channel in this dataset). It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Prostate central gland: 0.917 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task05_Prostate/prostate_central_gland Model input and output: ▸ Input: 1 channel MRI image (T2) ▸ Output: 2 channels for background & foreground

Annotation model	Description
‣ annotation_prostate_wg	A pre-trained model for volumetric (3D) annotation of the prostate whole gland (including peripheral zone) from the T2 MRI (we ignore the ADC channel in this dataset). It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ‣ Prostate central gland: 0.917 Model folder inside the docker: ‣ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task05_Prostate/prostate_whole_gland Model input and output: ‣ Input: 1 channel MRI image (T2) ‣ Output: 2 channels for background & foreground
Left atrium annotation	
‣ annotation_left_atrium	A pre-trained model for volumetric (3D) annotation of the left atrium from Mono-modal MRI. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ‣ Left atrium: 0.924 Model folder inside the docker: ‣ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task02_Heart

Annotation model	Description
	Model input and output: ▸ Input: 1 channel MRI image ▸ Output: 2 channels for background & foreground
Pancreas	
▸ annotation_pancreas	A pre-trained model for volumetric (3D) annotation of the pancreas (including tumors) from portal venous phase CT. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Pancreas (including tumor): 0.839 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task07_Pancreas Model input and output: ▸ Input: 1 channel CTimage ▸ Output: 2 channels for background & foreground
Colon tumor annotation	
▸ annotation_colon	A pre-trained model for volumetric (3D) annotation of the colon tumor primaries from CT image. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Colon tumor: 0.781

Annotation model	Description
	Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task10_Colon Model input and output: ▸ Input: 1 channel CTimage ▸ Output: 2 channels for background & foreground
Hepatic vessel annotation	
▸ annotation_hepaticvessel	A pre-trained model for volumetric (3D) annotation of the hepatic vessel and tumor from CT image. It is trained using our AIAA 3D model (dextr3D) 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 achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Hepatic vessel: 0.568 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/dextr3D/tasks/MSD_Task08_HepaticVessel Model input and output: ▸ Input: 1 channel CTimage ▸ Output: 2 channels for background & foreground
Spleen annotation	
▸ annotation_spleen	A pre-trained model for volumetric (3D) annotation of the spleen from CT image. It is trained using our AIAA 3D model (dextr3D) 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.

Annotation model	Description
	This model achieve the following Dice score on the validation data (our own split from the training dataset): ▸ Spleen: 0.963 Model folder inside the docker: ▸ /opt/nvidia/medical/annotation/ dextr3D/tasks/MSD_Task09_Spleen Model input and output: ▸ Input: 1 channel CTimage ▸ Output: 2 channels for background & foreground

10.2. 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:

Transforms	Description
	- 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". - 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*

Transforms	Description
	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. ▸ 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].

Transforms	Description
	- 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: - 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"].

Transforms	Description
	▸ 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 - 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*

Transforms	Description
	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* 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*

Transforms	Description
	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* 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

Transforms	Description
	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* "label". - pad: intnumber of voxels for adding a margin around the object) - foreground_only: booleanwhether to treat all foreground labels as one binary label (default) or whether to select foreground label at random.
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*

Transforms	Description
	"label". - sigma: *float* size of Gaussian. - pert: *float* maximum value of 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". ▸ 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*

Transforms	Description
	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* 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*

Transforms	Description
	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. ▸ 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)

www.nvidia.com