DeepReg

DeepReg

Oct 22, 2020

GETTING STARTED

Note: DeepReg [beta] will be released in Autumn 2020 - however, many tutorials, demos and much of the core functionality are already accessible.

DeepReg is a freely available, community-supported open-source toolkit for research and education in medical image registration using deep learning.

The current version is implemented as a TensorFlow2-based framework, and contains implementations for unsupervised- and weakly-supervised algorithms with their combinations and variants. DeepReg has a practical focus on growing and diverse clinical applications, as seen in the provided examples - DeepReg Demos.

Get involved and help make DeepReg better!

CHAPTER

ONE

FEATURES

DeepReg extends and simplifies workflows for medical imaging researchers working in TensorFlow 2, and can be easily installed and used for efficient training and rapid deployment of deep-learning registration algorithms.

DeepReg is designed to be used with minimal programming or scripting, owing to its built-in command line tools.

Our development and all related work involved in the project is public, and released under the Apache 2.0 license.

CHAPTER

TWO

CONTRIBUTORS

DeepReg is maintained and led by a team of developers and researchers. People with significant contributions to DeepReg are listed below (in alphabetical order).

Name	Affiliation (at time of contribution)
Adria Casamitjana	University College London
Alexander Grimwood	University College London
Ester Bonmati	University College London
Matt Clarkson	University College London
Nina Montana Brown	University College London
Qianye Yang	University College London
Remi Delaunay	University College London / King's College London
Shaheer Saaed	University College London
Tom Vercauteren	King's College London
Yipeng Hu	University College London
Yunguan Fu	University College London / InstaDeep
Zachary Baum	University College London
Zhe Min	University College London

This open-source initiative started within University College London, with support from the Wellcome/EPSRC Centre for Interventional and Surgical Sciences (WEISS), and partial support from the Wellcome/EPSRC Centre for Medical Engineering (CME).

CHAPTER

THREE

CONTACT

For development matters, please raise an issue.

For matters regarding the Code of Conduct, such as a complaint, please email the DeepReg Development Team: DeepRegNet@gmail.com.

Alternatively, please contact one or more members of the CoC Committee as appropriate: Nina Montana Brown (nina.brown.15@ucl.ac.uk), Ester Bonmati (e.bonmati@ucl.ac.uk), Matt Clarkson (m.clarkson@ucl.ac.uk).

3.1 Installation

DeepReg is written in Python 3.7. Dependent external libraries provide core IO functionalities and other standard processing tools. The dependencies for DeepReg are managed by `pip`.

3.1.1 Create a virtual environment

The recommended method is to install DeepReg in a dedicated virtual environment to avoid issues with other dependencies. The conda environment is recommended:

Anaconda / Miniconda

DeepReg primarily supports and is regularly tested with Ubuntu and Debian Linux distributions.

Linux

Mac OS

Windows

With CPU only

```
conda env create -f environment_cpu.yml
conda activate deepreg
```

With GPU

```
conda env create -f environment.yml
conda activate deepreg
```

With CPU only

```
conda env create -f environment_cpu.yml
conda activate deepreg
```

With GPU

Warning: Not supported or tested.

With CPU only

Warning: DeepReg on Windows is not fully supported. However, you can use the Windows Subsystem for Linux with CPU only. Set up WSL and follow the DeepReg setup instructions for Linux.

With GPU

Warning: Not supported or tested.

3.1.2 Use docker

We also provide the docker file for building the docker image.

Install docker

Docker can be installed following the official documentation.

For Linux based OS, there are some additional setup after the installation. Otherwise you might have permission errors.

Build docker image

```
docker build . -t deepreg -f Dockerfile
```

where

- `-t` names the built image as `deepreg`.
- `-f` provides the docker file for configuration.

Create a container

```
docker run --name <container_name> --privileged=true -ti deepreg bash
```

where - `--name` names the created container. - `--privileged=true` is required to solve the permission issue linked to TensorFlow profiler. - `-it` allows interaction with container and enters the container directly, check more info on stackoverflow.

Remove a container

```
docker rm -v <container_name>
```

which removes a created container and its volumes, check more info on docker documentation.

3.1.3 Install the package directly

Install from the cloned local project

```
git clone https://github.com/DeepRegNet/DeepReg.git # clone the repository
cd DeepReg # change working directory to the DeepReg root directory
pip install -e . # install the package
```

Install from the PyPI release

```
pip install deepreg
```

Note

All dependencies, APIs and command-line tools will be installed automatically via either installation method. However, the PyPI release currently does not ship with test data and demos. Running examples in this documentation may require downloading test data and changing default paths to user-installed packages with the PyPI release. These examples include those in the Quick Start and DeepReg Demo.

3.2 Quick Start

This is a set of simple tests to use DeepReg command line tools. More details and other options can be found in Command Line Tools.

First, install DeepReg and change current directory to the root directory of DeepReg.

3.2.1 Train a registration network

Train a registration network using unpaired and labeled example data with a predefined configuration:

```
deepreg_train --gpu "" --config_path config/unpaired_labeled_ddf.yaml --log_dir test
```

where:

- `--gpu ""` indicates using CPU. Change to `--gpu "0"` to use the GPU at index 0.
- `--config_path <filepath>` specifies the configuration file path.
- `--log_dir test` specifies the output folder. In this case, the output is saved in `logs/test`.

3.2.2 Evaluate a trained network

Once trained, evaluate the network using a test dataset:

```
deepreg_predict --gpu "" --ckpt_path logs/test/save/weights-epoch2.ckpt --mode test
```

where:

- `--ckpt_path <filepath>` specifies the checkpoint file path.
- `--mode test` specifies prediction on the test dataset.

3.2.3 Warp an image

DeepReg provides a command line interface (CLI) tool to warp an image/label with a dense displacement field (DDF):

```
deepreg_warp --image data/test/nifti/unit_test/moving_image.nii.gz --ddf data/test/
↪nifti/unit_test/ddf.nii.gz --out logs/test_warp/out.nii.gz
```

where:

- `--image <filepath>` specifies the image/label file path.
- `--ddf <filepath>` specifies the ddf file path.
- `--out <filepath>` specifies the output file path.

3.3 Image Registration with Deep Learning

A series of scientific tutorials on deep learning for registration can be found at the learn2reg tutorial, held in conjunction with MICCAI 2019.

This document provides a practical overview for a number of algorithms supported by DeepReg.

3.3.1 Registration

Image registration is the process of mapping the coordinate system of one image into another image. A registration method takes a pair of images as input, denoted as moving and fixed images. In this tutorial, we register the moving image into the fixed image, i.e. mapping the coordinates of the moving image onto the fixed image.

3.3.2 Network

Predict a dense displacement field

With deep learning, given a pair of moving and fixed images, the registration network outputs a dense displacement field (DDF) with the same shape as the moving image. Each value can be considered as the placement of the corresponding pixel / voxel of the moving image. Therefore, the DDF defines a mapping from the moving image's coordinates to the fixed image.

In this tutorial, we mainly focus on DDF-based methods.

Predict a dense velocity field

Another option is to predict a dense (static) velocity field (DVF), such that a diffeomorphic DDF can be numerically integrated. Read "A fast diffeomorphic image registration algorithm" and "Diffeomorphic demons: Efficient non-parametric image registration" for more details.

Predict an affine transformation

A more constrained option is to predict an affine transformation, parameterised by the affine transformation matrix of 12 degrees of freedom. The DDF can then be computed to resample the moving images in fixed image space.

Predict a region of interest

Instead of outputting the transformation between coordinates, given moving image, fixed image, and a region of interest (ROI) in the moving image, the network can predict the ROI in the fixed image directly. Interested readers are referred to the MICCAI 2019 paper: Conditional segmentation in lieu of image registration

3.3.3 Loss

A loss function has to be defined to train a deep neural network. There are mainly three types of losses:

Intensity based (image based) loss

This type of loss measures the dissimilarity of the fixed image and warped moving image, which is adapted from the classical image registration methods. Intensity based loss is modality-independent and similar to many other well-studied computer vision and medical imaging tasks, such as image segmentation.

The common loss functions are normalized cross correlation (NCC), sum of squared distance (SSD), and normalized mutual information (MI).

Feature based (label based) loss

This type of loss measures the dissimilarity of the fixed image labels and warped moving image labels. The label is often an ROI in the image, like the segmentation of an organ in a CT image.

The common loss function is Dice loss, Jacard and average cross-entropy over all voxels.

Deformation loss

This type of loss measures the deformation to regularize the transformation.

For DDF, the common loss functions are bending energy, L1 or L2 norm of the displacement gradient.

3.3.4 Learning

Depending on the availability of the data labels, registration networks can be trained with different approaches:

Unsupervised

When the data label is unavailable, the training can be driven by the unsupervised loss. The loss function often consists of the intensity based loss and deformation loss. The following is an illustration of an unsupervised DDF-based registration network.

Weakly-supervised

When there is no intensity based loss that is appropriate for the image pair one would like to register, the training can take a pair of corresponding moving and fixed labels (in addition to the image pair), represented by binary masks, to compute a label dissimilarity (feature based loss) to drive the registration.

Combined with the regularisation on the predicted displacement field, this forms a weakly-supervised training. An illustration of an weakly-supervised DDF-based registration network is provided below.

When multiple labels are available for each image, the labels can be sampled during the training iteration, such that only one label per image is used in each iteration of the data set (epoch).

Combined

When the data label is available, combining intensity based, feature based, and deformation based losses together has shown superior registration accuracy, compared to unsupervised and weakly supervised methods. Following is an illustration of a combined DDF-based registration network.

3.4 Design Experiments

DeepReg dataset loaders use a folder/directory-based file storing approach, with which the user will be responsible for organising image and label files in required file formats and folders. This design was primarily motivated by the need to minimise the risk of data leakage (or information leakage), both in code development and subsequent applications.

3.4.1 Random-split

Every call of the `deepreg_train` or `deepreg_predict` function uses a dataset "physically" separated by folders, including 'train', 'val' and 'test' sets used in a random-split experiment. In this case, the user needs to randomly assign available experiment image and label files into the three folders. Again, for more details see the Dataset loader.

3.4.2 Cross-validation

Experiments such as *cross-validation* can be readily implemented by using the "multi-folder support" in the `dataset` section of the yaml configuration files. See details in configuration.

For example, in a 3-fold cross-validation, the user may randomly partition available experiment data files into four folders, 'fold0', 'fold1', 'fold2' and 'test'. The 'test' is a hold-out testing set. Each run of the 3-fold cross-validation then can be specified in a different yaml file as follows.

"cv_run1.yaml":

```yaml
dataset:
  dir:
    train: # training data set
      - "data/test/h5/paired/fold0"
      - "data/test/h5/paired/fold1"
    valid: "data/test/h5/paired/fold2" # validation data set
    test: ""
```

"cv_run2.yaml":

```yaml
dataset:
  dir:
    train: # training data set
      - "data/test/h5/paired/fold0"
      - "data/test/h5/paired/fold2"
    valid: "data/test/h5/paired/fold1" # validation data set
    test: ""
```

"cv_run3.yaml":

```yaml
dataset:
  dir:
    train: # training data set
      - "data/test/h5/paired/fold1"
      - "data/test/h5/paired/fold2"
    valid: "data/test/h5/paired/fold0" # validation data set
    test: ""
```

To further facilitate flexible uses of these dataset loaders, the `deepreg_train` and `deepreg_predict` functions also accept multiple yaml files - therefore the same `train` section does not have to be repeated multiple times for the multiple cross-validation folds or for the test. An example `dataset` section for configuring testing when using `deepreg_predict` is given below.

"test.yaml":

```yaml
dataset:
  dir:
    train: ""
    valid: ""
    test: "data/test/h5/paired/test" # validation data set
```

3.5 Custom Functionalities

Besides the implemented features provided in DeepReg, we provide the following tutorials for implementing custom functionalities.

- Custom loss function
- Custom network

3.5.1 Custom loss function

This tutorial will take an image intensity based loss function (mutual information) as an example to show how to add a new loss to DeepReg.

A brief review of the types of loss functions in DeepReg

Three main types of the loss functions are supported in DeepReg: `intensity (image) based loss`, `label based loss` and `deformation loss`. See Docs here for details. The corresponding source files for the losses is included in `deepreg/model/loss`.

Step 1: Add the new function in loss source code

The first step is to add your own loss function, which should take at least 2 parameters, `y_true` for the ground truth and `y_pred` for the prediction. e.g. in `deepreg/model/loss/image.py`. The loss can be defined as:

```
def global_mutual_information(y_true: tf.Tensor, y_pred: tf.Tensor) -> tf.Tensor:
    """
    differentiable global mutual information loss via Parzen windowing method.
    reference: https://dspace.mit.edu/handle/1721.1/123142, Section 3.1, equation 3.1-
↪3.5
    :y_true: shape = (batch, dim1, dim2, dim3, ch)
    :y_pred: shape = (batch, dim1, dim2, dim3, ch)
    :return: shape = (batch,)
    """
    ...
    return tf.reduce_sum(pab * tf.math.log(pab / papb + eps), axis=[1, 2])
```

In order to be compatible with the pipeline in DeepReg, another modification is needed in `deeepreg/model/loss/image.py`. The modification is simply adding an `elif` branch to the `dissimilarity_fn` function. The following code block shows the added `elif` branch where we use `"gmi"` to represent the global mutual information:

```
def dissimilarity_fn(
    y_true: tf.Tensor, y_pred: tf.Tensor, name: str, **kwargs
) -> tf.Tensor:
    """
    :param y_true: fixed_image, shape = (batch, f_dim1, f_dim2, f_dim3)
    :param y_pred: warped_moving_image, shape = (batch, f_dim1, f_dim2, f_dim3)
    :param name: name of the dissimilarity function
    :param kwargs: absorb additional parameters
    :return: shape = (batch,)
    """
    assert name in ["lncc", "ssd", "gmi"]
    # shape = (batch, f_dim1, f_dim2, f_dim3, 1)
    y_true = tf.expand_dims(y_true, axis=4)
```

(continues on next page)

(continued from previous page)

```
    y_pred = tf.expand_dims(y_pred, axis=4)
    if name == "lncc":
        return -local_normalized_cross_correlation(y_true, y_pred, **kwargs)
    elif name == "ssd":
        return ssd(y_true, y_pred)
    elif name == "gmi":
        return -global_mutual_information(y_true, y_pred)
    else:
        raise ValueError("Unknown loss type.")
```

Step 2: Add test functions (for contributing developers, optional for users)

Add corresponding unit test for the new added functions to `deepreg/test/unit`. This is optional for the users. Everyone is warmly welcome to make contribution to DeepReg. Please follow our contribution guidelines here.

Step 3: Set yaml configuration files

We take the paired prostate MR and Ultrasound registration demo as an example. In order to use the newly added loss, all that is needed is to modify the loss configuration in the train configuration file `paired_mrus_prostate_train.yaml` (lines 10-15):

```
# define the loss function for training
  loss:
    dissimilarity:
      image:
        name: "gmi"
        weight: 1.0
```

That's it. Follow the instructions in the demo to begin training with the newly added loss.

3.5.2 Custom network

This tutorial shows how to define a new network and add it to DeepReg, using a specific example for adding a GlobalNet to predict an affine transformation, as opposed to nonrigid transformation.

For general guidance on making a contribution to DeepReg, see the contribution guidelines.

Step 1: Create network backbone

The first step is to create a new backbone class, which consists of the neural network architecture you want to use, and place it in the backbone directory `deepreg/model/backbone/`. The affine method uses the GlobalNet network architecture (`deepreg/model/backbone/global_net.py`) from Hu et al. 2018. The GlobalNet network has an encoder-only architecture, which is used to predict the parameters of an affine transformation model, with 12 degrees of freedom.

We recommend using the tf.keras API to write your network, by defining the layers of your backbone class in `def __init__()` and the network's forward pass in `def call()`. Custom DeepReg layers can be found in `deepreg/model/layer.py`.

```
class GlobalNet(tf.keras.Model):
    """
    Builds GlobalNet for image registration based on
    Y. Hu et al.,
    "Label-driven weakly-supervised learning for multimodal
    deformable image registration,"
    (ISBI 2018), pp. 1070-1074.
    """

   def __init__(
        self,
        image_size,
        out_channels,
        num_channel_initial,
        extract_levels,
        out_kernel_initializer,
        out_activation,
        **kwargs,
    ):
        """
        Image is encoded gradually, i from level 0 to E.
        Then, a densely-connected layer outputs an affine
        transformation.
        :param out_channels: int, number of channels for the output
        :param num_channel_initial: int, number of initial channels
        :param extract_levels: list, which levels from net to extract
        :param out_activation: str, activation at last layer
        :param out_kernel_initializer: str, which kernel to use as initialiser
        :param kwargs:
        """
        super(GlobalNet, self).__init__(**kwargs)
        # save parameters
        self._extract_levels = extract_levels
        self._extract_max_level = max(self._extract_levels)  # E
        self.reference_grid = layer_util.get_reference_grid(image_size)
        self.transform_initial = tf.constant_initializer(
            value=[1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
        )
        # init layer variables
        num_channels = [
            num_channel_initial * (2 ** level)
            for level in range(self._extract_max_level + 1)
        ]  # level 0 to E
        self._downsample_blocks = [
            layer.DownSampleResnetBlock(
                filters=num_channels[i], kernel_size=7 if i == 0 else 3
            )
            for i in range(self._extract_max_level)
        ]  # level 0 to E-1
        self._conv3d_block = layer.Conv3dBlock(filters=num_channels[-1])  # level E
        self._dense_layer = layer.Dense(
            units=12, bias_initializer=self.transform_initial
        )

    def call(self, inputs, training=None, mask=None):
        """
        Build GlobalNet graph based on built layers.
```

(continues on next page)

(continued from previous page)

```
        :param inputs: image batch, shape = [batch, f_dim1, f_dim2, f_dim3, ch]
        :param training:
        :param mask:
        :return:
        """
        # down sample from level 0 to E
        h_in = inputs
        for level in range(self._extract_max_level):  # level 0 to E - 1
            h_in, _ = self._downsample_blocks[level](inputs=h_in, training=training)
        h_out = self._conv3d_block(
            inputs=h_in, training=training
        )  # level E of encoding
        # predict affine parameters theta of shape = [batch, 4, 3]
        self.theta = self._dense_layer(h_out)
        self.theta = tf.reshape(self.theta, shape=(-1, 4, 3))
        # warp the reference grid with affine parameters to output a ddf
        grid_warped = layer_util.warp_grid(self.reference_grid, self.theta)
        output = grid_warped - self.reference_grid
        return output`
```

In order to use the backbone network in the DeepReg pipeline, a new option needs to be added to `build_backbone()` from `deepreg/model/network/util.py`. We use the keyword "global" here to refer to our `GlobalNet` class and `"affine"` for the method name. This will allow us to define the backbone network directly in the configuration file.

```
def build_backbone(
    image_size: tuple, out_channels: int, model_config: dict, method_name: str
) -> tf.keras.Model:
    """
    Backbone model accepts a single input of shape (batch, dim1, dim2, dim3, ch_in)
    and returns a single output of shape (batch, dim1, dim2, dim3, ch_out)
    :param image_size: tuple, dims of image, (dim1, dim2, dim3)
    :param out_channels: int, number of out channels, ch_out
    :param method_name: str, one of ddf | dvf | conditional
    :param model_config: dict, model configuration, returned from parser.yaml.load
    :return: tf.keras.Model
    """
    if not (
        (isinstance(image_size, tuple) or isinstance(image_size, list))
        and len(image_size) == 3
    ):
        raise ValueError(f"image_size must be tuple of length 3, got {image_size}")
    if not (isinstance(out_channels, int) and out_channels >= 1):
        raise ValueError(f"out_channels must be int >=1, got {out_channels}")
    if not (isinstance(model_config, dict) and "backbone" in model_config.keys()):
        raise ValueError(
            f"model_config must be a dict having key 'backbone', got{model_config}"
        )
    if method_name not in ["ddf", "dvf", "conditional", "affine"]:
        raise ValueError(
            "method name has to be one of ddf/dvf/conditional/affine in build_
↪backbone, "
            "got {}".format(method_name)
        )

    if method_name in ["ddf", "dvf"]:
```

(continues on next page)

(continued from previous page)

```
        out_activation = None
        # TODO try random init with smaller number
        out_kernel_initializer = "zeros"  # to ensure small ddf and dvf
    elif method_name in ["conditional"]:
        out_activation = "sigmoid"  # output is probability
        out_kernel_initializer = "glorot_uniform"
    elif method_name in ["affine"]:
        out_activation = None
        out_kernel_initializer = "zeros"
    else:
        raise ValueError("Unknown method name {}".format(method_name))

    if model_config["backbone"] == "local":
        return LocalNet(
            image_size=image_size,
            out_channels=out_channels,
            out_kernel_initializer=out_kernel_initializer,
            out_activation=out_activation,
            **model_config["local"],
        )
    elif model_config["backbone"] == "global":
        return GlobalNet(
            image_size=image_size,
            out_channels=out_channels,
            out_kernel_initializer=out_kernel_initializer,
            out_activation=out_activation,
            **model_config["global"],
        )
    elif model_config["backbone"] == "unet":
        return UNet(
            image_size=image_size,
            out_channels=out_channels,
            out_kernel_initializer=out_kernel_initializer,
            out_activation=out_activation,
            **model_config["unet"],
        )
    else:
        raise ValueError("Unknown model name")
```

Step 2: Create network model

We can now create a network model for the affine method in `deepreg/model/network/affine.py` . We first need to write the model's forward pass, which makes use of the backbone network class to predict an affine transformation which will be used to output a dense displacement field (DDF).

```
def affine_forward(
    backbone: tf.keras.Model,
    moving_image: tf.Tensor,
    fixed_image: tf.Tensor,
    moving_label: (tf.Tensor, None),
    moving_image_size: tuple,
    fixed_image_size: tuple,
):
    """
    Perform the network forward pass
```

(continues on next page)

(continued from previous page)

```
    :param backbone: model architecture object, e.g. model.backbone.local_net
    :param moving_image: tensor of shape (batch, m_dim1, m_dim2, m_dim3)
    :param fixed_image:  tensor of shape (batch, f_dim1, f_dim2, f_dim3)
    :param moving_label: tensor of shape (batch, m_dim1, m_dim2, m_dim3) or None
    :param moving_image_size: tuple like (m_dim1, m_dim2, m_dim3)
    :param fixed_image_size: tuple like (f_dim1, f_dim2, f_dim3)
    :return: tuple(_affine, _ddf, _pred_fixed_image, _pred_fixed_label)
    :return: tuple(affine, ddf, pred_fixed_image, pred_fixed_label, fixed_grid), where
    - affine is the affine transformation matrix predicted by the network (batch, 4,␣
↪3)
    - ddf is the dense displacement field of shape (batch, f_dim1, f_dim2, f_dim3, 3)
    - pred_fixed_image is the predicted (warped) moving image of shape (batch, f_dim1,
↪ f_dim2, f_dim3)
    - pred_fixed_label is the predicted (warped) moving label of shape (batch, f_dim1,
↪ f_dim2, f_dim3)
    - fixed_grid is the grid of shape(f_dim1, f_dim2, f_dim3, 3)
    """

    # expand dims
    # need to be squeezed later for warping
    moving_image = tf.expand_dims(
        moving_image, axis=4
    )  # (batch, m_dim1, m_dim2, m_dim3, 1)
    fixed_image = tf.expand_dims(
        fixed_image, axis=4
    )  # (batch, f_dim1, f_dim2, f_dim3, 1)

    # adjust moving image
    moving_image = layer_util.resize3d(
        image=moving_image, size=fixed_image_size
    )  # (batch, f_dim1, f_dim2, f_dim3, 1)

    # ddf, dvf
    inputs = tf.concat(
        [moving_image, fixed_image], axis=4
    )  # (batch, f_dim1, f_dim2, f_dim3, 2)
    ddf = backbone(inputs=inputs)  # (batch, f_dim1, f_dim2, f_dim3, 3)
    affine = backbone.theta

    # prediction, (batch, f_dim1, f_dim2, f_dim3)
    warping = layer.Warping(fixed_image_size=fixed_image_size)
    grid_fixed = tf.squeeze(warping.grid_ref, axis=0)  # (f_dim1, f_dim2, f_dim3, 3)
    pred_fixed_image = warping(inputs=[ddf, tf.squeeze(moving_image, axis=4)])
    pred_fixed_label = (
        warping(inputs=[ddf, moving_label]) if moving_label is not None else None
    )
    return affine, ddf, pred_fixed_image, pred_fixed_label, grid_fixed
```

Similar to `build_backbone` we also need to write the `build_affine_model` function, which consists of building the model according to the networks' inputs, backbone and loss function.

```
def build_affine_model(
    moving_image_size: tuple,
    fixed_image_size: tuple,
    index_size: int,
    labeled: bool,
```

(continues on next page)

(continued from previous page)

```
    batch_size: int,
    model_config: dict,
    loss_config: dict,
    ):
    """
    :param moving_image_size: (m_dim1, m_dim2, m_dim3)
    :param fixed_image_size: (f_dim1, f_dim2, f_dim3)
    :param index_size: int, the number of indices for identifying a sample
    :param labeled: bool, indicating if the data is labeled
    :param batch_size: int, size of mini-batch
    :param model_config: config for the model
    :param loss_config: config for the loss
    :return: the built tf.keras.Model
    """

    # inputs
    (moving_image, fixed_image, moving_label, fixed_label, indices) = build_inputs(
        moving_image_size=moving_image_size,
        fixed_image_size=fixed_image_size,
        index_size=index_size,
        batch_size=batch_size,
        labeled=labeled,
    )

    # backbone
    backbone = build_backbone(
        image_size=fixed_image_size,
        out_channels=3,
        model_config=model_config,
        method_name=model_config["method"],
    )

    # forward
    affine, ddf, pred_fixed_image, pred_fixed_label, grid_fixed = affine_forward(
        backbone=backbone,
        moving_image=moving_image,
        fixed_image=fixed_image,
        moving_label=moving_label,
        moving_image_size=moving_image_size,
        fixed_image_size=fixed_image_size,
    )

    # build model
    inputs = {
        "moving_image": moving_image,
        "fixed_image": fixed_image,
        "indices": indices,
    }
    outputs = {"ddf": ddf, "affine": affine}
    model_name = model_config["method"].upper() + "RegistrationModel"
    if moving_label is None:  # unlabeled
        model = tf.keras.Model(
            inputs=inputs, outputs=outputs, name=model_name + "WithoutLabel"
        )
    else:  # labeled
        inputs["moving_label"] = moving_label
        inputs["fixed_label"] = fixed_label
```

(continues on next page)

(continued from previous page)

```
        outputs["pred_fixed_label"] = pred_fixed_label
        model = tf.keras.Model(
            inputs=inputs, outputs=outputs, name=model_name + "WithLabel"
        )

    # add loss and metric
    model = add_ddf_loss(model=model, ddf=ddf, loss_config=loss_config)
    model = add_image_loss(
        model=model,
        fixed_image=fixed_image,
        pred_fixed_image=pred_fixed_image,
        loss_config=loss_config,
    )
    model = add_label_loss(
        model=model,
        grid_fixed=grid_fixed,
        fixed_label=fixed_label,
        pred_fixed_label=pred_fixed_label,
        loss_config=loss_config,
    )

    return model
```

Finally, the last step consists of adding the `build_affine_model` option to `deepreg/model/network/build.py` to be able to parse it from the configuration file.

```
def build_model(
    moving_image_size: tuple,
    fixed_image_size: tuple,
    index_size: int,
    labeled: bool,
    batch_size: int,
    model_config: dict,
    loss_config: dict,
):
    """
    Parsing algorithm types to model building functions
    :param moving_image_size: [m_dim1, m_dim2, m_dim3]
    :param fixed_image_size: [f_dim1, f_dim2, f_dim3]
    :param index_size: dataset size
    :param labeled: true if the label of moving/fixed images are provided
    :param batch_size: mini-batch size
    :param model_config: model configuration, e.g. dictionary return from parser.yaml.
↪load
    :param loss_config: loss configuration, e.g. dictionary return from parser.yaml.
↪load
    :return: the built tf.keras.Model
    """
    if model_config["method"] in ["ddf", "dvf"]:
        return build_ddf_dvf_model(
            moving_image_size=moving_image_size,
            fixed_image_size=fixed_image_size,
            index_size=index_size,
            labeled=labeled,
            batch_size=batch_size,
            model_config=model_config,
```

(continues on next page)

(continued from previous page)

```
            loss_config=loss_config,
        )
    elif model_config["method"] == "conditional":
        return build_conditional_model(
            moving_image_size=moving_image_size,
            fixed_image_size=fixed_image_size,
            index_size=index_size,
            labeled=labeled,
            batch_size=batch_size,
            model_config=model_config,
            loss_config=loss_config,
        )
    elif model_config["method"] == "affine":
        return build_affine_model(
            moving_image_size=moving_image_size,
            fixed_image_size=fixed_image_size,
            index_size=index_size,
            labeled=labeled,
            batch_size=batch_size,
            model_config=model_config,
            loss_config=loss_config,
        )
    else:
        raise ValueError("Unknown model method")
```

Step 3: Testing (for contributing developers, optional for users)

Everyone is warmly welcome to make contributions to DeepReg and add corresponding unit test for the newly added functions to `test/unit/`. Recommendations regarding testing style can be found in the contribution guidelines. Here is a practical example of unit tests made for our affine model in `test/unit/test_affine.py`:

```
def test_affine_forward():
    """
    Testing that affine_forward function returns the tensors with correct shapes
    """

    moving_image_size = (1, 3, 5)
    fixed_image_size = (2, 4, 6)
    batch_size = 1

    global_net = build_backbone(
        image_size=fixed_image_size,
        out_channels=3,
        model_config={
            "backbone": "global",
            "global": {"num_channel_initial": 4, "extract_levels": [1, 2, 3]},
        },
        method_name="affine",
    )

    # Check conditional mode network output shapes - Pass
    affine, ddf, pred_fixed_image, pred_fixed_label, grid_fixed = affine_forward(
        backbone=global_net,
        moving_image=tf.ones((batch_size,) + moving_image_size),
        fixed_image=tf.ones((batch_size,) + fixed_image_size),
```

(continues on next page)

(continued from previous page)

```
        moving_label=tf.ones((batch_size,) + moving_image_size),
        moving_image_size=moving_image_size,
        fixed_image_size=fixed_image_size,
    )
    assert affine.shape == (batch_size,) + (4,) + (3,)
    assert ddf.shape == (batch_size,) + fixed_image_size + (3,)
    assert pred_fixed_image.shape == (batch_size,) + fixed_image_size
    assert pred_fixed_label.shape == (batch_size,) + fixed_image_size
    assert grid_fixed.shape == fixed_image_size + (3,)


def test_build_affine_model():
    """
    Testing that build_affine_model function returns the tensors with correct shapes
    """
    moving_image_size = (1, 3, 5)
    fixed_image_size = (2, 4, 6)
    batch_size = 1

    model = build_affine_model(
        moving_image_size=moving_image_size,
        fixed_image_size=fixed_image_size,
        index_size=1,
        labeled=True,
        batch_size=batch_size,
        model_config={
            "method": "affine",
            "backbone": "global",
            "global": {"num_channel_initial": 4, "extract_levels": [1, 2, 3]},
        },
        loss_config={
            "dissimilarity": {
                "image": {"name": "lncc", "weight": 0.1},
                "label": {
                    "name": "multi_scale",
                    "weight": 1,
                    "multi_scale": {
                        "loss_type": "dice",
                        "loss_scales": [0, 1, 2, 4, 8, 16, 32],
                    },
                },
            },
            "regularization": {"weight": 0.0, "energy_type": "bending"},
        },
    )

    inputs = {
        "moving_image": tf.ones((batch_size,) + moving_image_size),
        "fixed_image": tf.ones((batch_size,) + fixed_image_size),
        "indices": 1,
        "moving_label": tf.ones((batch_size,) + moving_image_size),
        "fixed_label": tf.ones((batch_size,) + fixed_image_size),
    }

    outputs = model(inputs)

    expected_outputs_keys = ["affine", "ddf", "pred_fixed_label"]
```

(continues on next page)

(continued from previous page)

```
    assert all(keys in expected_outputs_keys for keys in outputs)
    assert outputs["pred_fixed_label"].shape == (batch_size,) + fixed_image_size
    assert outputs["affine"].shape == (batch_size,) + (4,) + (3,)
    assert outputs["ddf"].shape == (batch_size,) + fixed_image_size + (3,)
```

Step 4: Set yaml configuration files

An example of yaml configuration file for the affine method is available in `config/unpaired_labeled_affine.yaml`. For using both the GlobalNet backbone and affine method you will need to add their aforementioned keyword "global" and "affine". Optional parameters such as `out_kernel_initializer` or `num_channel_initial` can also be specified. A snippet of `config/unpaired_labeled_affine.yaml` is shown below. Please see the configuration documentation for more details.

```
model:
method: "affine"
backbone:
  name: "global"
  out_kernel_initializer: "zeros"
  out_activation: ""
global:
  num_channel_initial: 1
  extract_levels: [0, 1, 2, 3, 4]
```

3.6 Introduction to DeepReg Demos

DeepReg offers multiple built-in dataset loaders to support real-world clinical scenarios, in which images may be paired, unpaired or grouped. Images may also be labeled with segmented regions of interest to assist registration.

A typical workflow to develop a registration network using DeepReg includes:

- Select a dataset loader, among the unpaired, paired and grouped, and prepare data into folders as required;
- Configure the network training in the configuration yaml file(s), as specified in supported configuration details;
- Train and tune the registration network with the command line tool `deepreg_train`;
- Test or use the final trained registration network with the command line tool `deepreg_predict`.

Besides the tutorials, a series of DeepReg Demos are provided to showcase a wide range of applications with real clinical image and label data. These applications range from ultrasound, CT and MR images, covering many clinical specialties such as neurology, urology, gastroenterology, oncology, respiratory and cardiovascular diseases.

Each DeepReg Demo provides a step-by-step instruction to explain how different scenarios can be implemented with DeepReg. All data sets used are open-accessible. Pre-trained models with numerical and graphical inference results are also available.

Note: DeepReg Demos are provided to demonstrate functionalities in DeepReg. Although effort has been made to ensure these demos are representative of real-world applications, the implementations and the results are not peer-reviewed or tested for clinical efficacy. Substantial further adaptation and development may be required for any potential clinical adoption.

3.7 Paired Images

The following DeepReg Demos provide examples of using paired images.

- Paired lung CT registration

  This demo registers paired CT lung images, with optional weak supervision.

- Paired brain MR-ultrasound registration

  This demo registers paired preoperative MR images and 3D tracked ultrasound images for locating brain tumours during neurosurgery, with optional weak supervision.

- Paired prostate MR-ultrasound registration

  This demo registers paired MR-to-ultrasound prostate images, an example of weakly-supervised multimodal image registration.

3.7.1 Paired lung CT registration

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

Author

DeepReg Development Team (Shaheer Saeed)

Application

This is a registration between CT images acquired at different time points for a single patient. The images being registered are taken at inspiration and expiration for each subject. This is an intra subject registration. This type of intra subject registration is useful when there is a need to track certain features on a medical image such as tumor location when conducting invasive procedures.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- The `demo_data.py`, `demo_train.py` and `demo_predict.py` scripts need to be run using the following command:

```
python3 demos/paired_ct_lung/script_name.py
```

A short description of the scripts is provided below. The scripts must be run in the following order:

- Run the `demo_data.py` script: This script does the following:
  - Download data using `tf.keras.utils.get_file`. Data is downloaded to the demo directory but this can be changed (instructions in the comments in the script).
  - Split the data into three sets train, valid and test (change ratio_of_test_and_valid_samples variable to change the ratio of test and valid samples)

  - Restructure the files, for each of the train, valid and test sets, into a directory structure that is suitable for use with the paired loader in DeepReg
  - Rescale all images to 0-255 so they are suitable for use with DeepReg
  - Download a pretrained model to use with the predict script
- Run the `demo_train.py` script: This script does the following:
  - Specify the training options like GPU support
  - Specify the config file paths (the config file to define the network is one which is avialable with DeepReg and the config file for the data is given in the demo folder)
  - Train a network using DeepReg
- Run the `demo_predict.py` script: This script does the following:
  - Use the pretrained network to make predictions for the test set
  - Use the predictions to plot the results (the path to the images generated in the logs will need to be specified, look at comments in the script to change this)

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/learn2reg_t2_paired_train_logs/.

Data

The dataset for this demo comes from Lean2Reg Challenge: CT Lung Registration - Training Data [1].

Tested DeepReg version

Last commit at which demo was tested: v. 0.1.6-alpha

Contact

Please raise an issue.

Reference

[1] Hering, Alessa, Murphy,Keelin, and van Ginneken, Bram. (2020). Lean2Reg Challenge: CT Lung Registration : CT Lung Registration - Training Data

3.7.2 Paired brain MR-ultrasound registration

Note: Please read the DeepReg Demo Disclaimer.

Source Code

Author

DeepReg Development Team (Shaheer Saeed)

Application

This demo aims to register pairs of brain MR and ultrasound scans. The dataset consists of 22 subjects with low-grade brain gliomas who underwent brain tumour resection [1]. The main application for this type of registration is to better delineate brain tumour boundaries during surgery and correct tissue shift induced by the craniotomy.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- The `demo_data.py`, `demo_train.py` and `demo_predict.py` scripts need to be run using the following command:

```
python3 demos/paired_mrus_brain/script_name.py
```

A short description of the scripts is provided below. The scripts must be run in the following order:

- Run the demo_data.py script: This script does the following:
  - Download a reduced copy of the dataset which has already been preprocessed
  - Download a pretrained model for use with the predict function
  - Note: This script can also be used to work with the full dataset by uncommenting the relevant sections in the script (please read the scripts' comments to see how to download the full dataset)
- Run the demo_train.py script: This script does the following:
  - Specify the training options like GPU support
  - Specify the config file paths (to define both the network config available in DeepReg and the data config given in the demo folder)
  - Train a network using DeepReg
- Run the demo_predict.py script: This script does the following:
  - Use the pretrained network to make predictions for the test set
  - Use the predicitions to plot the results (the images path generated in the logs will need to be specified)
- Note: The number of epochs and reduced dataset size for training will result in a loss in test accuracy so please train with the full dataset and for a greater number of epochs for improved results.

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/learn2reg_t1_paired_train_logs/.

Data

The dataset for this demo comes from Xiao et al. [1] and can be downloaded from:

https://archive.sigma2.no/pages/public/datasetDetail.jsf?id=10.11582/2020.00025

Tested DeepReg version

Last commit at which demo was tested: c709a46c345552ae1396e6d7ba46a44f7950aea0

Contact

Please raise an issue.

Reference

[1] Y. Xiao, M. Fortin, G. Unsgård , H. Rivaz, and I. Reinertsen, "REtroSpective Evaluation of Cerebral Tumors (RESECT): a clinical database of pre-operative MRI and intra-operative ultrasound in low-grade glioma surgeries". Medical Physics, Vol. 44(7), pp. 3875-3882, 2017.

3.7.3 Paired prostate MR-ultrasound registration

Note: Please read the DeepReg Demo Disclaimer.

Source Code

This demo uses DeepReg to re-implament the algorithms described in Weakly-supervised convolutional neural networks for multimodal image registration. A standalone demo was hosted at https://github.com/yipenghu/label-reg.

Author

DeepReg Development Team

Application

Registering preoperative MR images to intraoperative transrectal ultrasound images has been an active research area for more than a decade. The multimodal image registration task assist a number of ultrasound-guided interventions and surgical procedures, such as targeted biopsy and focal therapy for prostate cancer patients. One of the key challenges in this registration task is the lack of robust and effective similarity measures between the two image types. This demo implements a weakly-supervised learning approach to learn voxel correspondence between intensity patterns between the multimodal data, driven by expert-defined anatomical landmarks, such as the prostate gland segmenation.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download 10 folds of unpaired 3D ultrasound images and the pre-trained model.

```
python demos/paired_mrus_prostate/demo_data.py
```

- Call `deepreg_train` from command line. The following example uses two GPUs and launches the first of the ten runs of a 9-fold cross-validation, as specified in the ``*dataset*` section <./paired_mrus_prostate_dataset0.yaml>`_ and the ``*train*` section <./paired_mrus_prostate_train.yaml>`_, which can be specified in separate yaml files;

```
deepreg_train --gpu "1, 2" --config_path demos/paired_mrus_prostate/paired_mrus_
↪prostate_dataset0.yaml demos/paired_mrus_prostate/paired_mrus_prostate_train.yaml --
↪log_dir paired_mrus_prostate
```

- Call `deepreg_predict` from command line to use the saved ckpt file for testing on the data partitions specified in the config file, a copy of which will be saved in the [log_dir]. The following example uses a pre-trained model, on CPU. If not specified, the results will be saved at the created timestamp-named directories under /logs.

```
deepreg_predict --gpu "" --config_path demos/paired_mrus_prostate/paired_mrus_
↪prostate_dataset0.yaml demos/paired_mrus_prostate/paired_mrus_prostate_train.yaml --
↪ckpt_path demos/paired_mrus_prostate/dataset/pre-trained/weights-epoch500.ckpt --
↪mode test
```

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/paired_mrus_prostate/.

Data

This is a demo without real clinical data due to regulatory restrictions. The MR and ultrasound images used are simulated dummy images.

Tested DeepReg version

Last commit at which demo was tested: 7ec0f5157a81cd5e60cadd61bd617b433039d0e6

Contact

Please raise an issue.

3.8 Unpaired Images

The following DeepReg Demos provide examples of using unpaired images.

- Unpaired abdominal CT registration

  This demo compares three training strategies, using unsupervised, weakly-supervised and combined losses, to register inter-subject abdominal CT images.

- Unpaired lung CT registration

  This demo registers unpaired CT lung images, with optional weak supervision.

- Unpaired hippocampus MR registration

  This demo aligns hippocampus on MR images between different patients, with optional weak supervision.

- Unpaired prostate ultrasound registration

  This demo registers 3D ultrasound images with a 9-fold cross-validation. This strategy is applicable for any of the available dataset loaders.

3.8.1 Unpaired abdomen CT registration

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

Author

DeepReg Development Team (Ester Bonmati)

Application

This demo shows how to register unpaired abdominal CT data from different patients using DeepReg. In addition, the demo demonstrates the difference between the unsupervised, weakly-supervised and their combination, using a U-Net.

Data

The data set is from the MICCAI Learn2Reg grand challenge (https://learn2reg.grand-challenge.org/) task 3 [1], and can be downloaded directly from https://learn2reg.grand-challenge.org/Datasets/.

Instruction

- Install DeepReg;
- Change the working directory to the root directory of DeepReg project;
- Run [demo_data.py] to download and extract all files, and to split the data into training, validation and testing. If the data has already been downloaded. This will also download the pre-trained models:

```
python ./demos/unpaired_ct_abdomen/demo_data.py
```

After running the command you will have the following directories in DeepReg/demos/unpaired_ct_abdomen/dataset:

```
pre-trained  test  train  val
```

- The next step is to train the network using DeepReg. To train the network, run one of the following commands in command line:
- Unsupervised learning

```
deepreg_train --gpu "0" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪unsup.yaml --log_dir unpaired_ct_abdomen_unsup
```

- Weakly-supervised learning

```
deepreg_train --gpu "1" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪weakly.yaml --log_dir unpaired_ct_abdomen_weakly
```

- Combined learning

```
deepreg_train --gpu "2" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪comb.yaml --log_dir unpaired_ct_abdomen_comb
```

- After training the network, run `demo_predict`:

The following example uses a pre-trained model, on CPU.

```
deepreg_predict --gpu "" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪unsup.yaml --ckpt_path demos/unpaired_ct_abdomen/dataset/pre-trained/unsup/weights-
↪epoch5000.ckpt --log_dir unpaired_ct_abdomen_unsup --save_png --mode test
```

```
deepreg_predict --gpu "" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪weakly.yaml --ckpt_path demos/unpaired_ct_abdomen/dataset/pre-trained/weakly/
↪weights-epoch2250.ckpt --log_dir unpaired_ct_abdomen_weakly --save_png --mode test
```

```
deepreg_predict --gpu "" --config_path demos/unpaired_ct_abdomen/unpaired_ct_abdomen_
↪comb.yaml --ckpt_path demos/unpaired_ct_abdomen/dataset/pre-trained/comb/weights-
↪epoch2000.ckpt --log_dir unpaired_ct_abdomen_comb --save_png --mode test
```

- Finally, prediction results can be seen in the respective test folders specified in `deepreg_predict`.

Pre-trained model

Three pre-trained models are available for this demo, for different training strategies described above. These will be downloaded in respective sub-folders under the /dataset folder using the `demo_data.py`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/unpaired_ct_abdomen_unsup/, /logs/unpaired_ct_abdomen_weakly/ or /logs/unpaired_ct_abdomen_comb/.

Tested DeepReg version

Last commit at which demo was tested: 3157f880eb99ce10fc3a4a8ebcc595bd67be24e1

Contact

Please raise an issue.

Reference

[1] Adrian Dalca, Yipeng Hu, Tom Vercauteren, Mattias Heinrich, Lasse Hansen, Marc Modat, Bob de Vos, Yiming Xiao, Hassan Rivaz, Matthieu Chabanas, Ingerid Reinertsen, Bennett Landman, Jorge Cardoso, Bram van Ginneken, Alessa Hering, and Keelin Murphy. (2020, March 19). Learn2Reg - The Challenge. Zenodo. http://doi.org/10.5281/zenodo.3715652

3.8.2 Unpaired lung CT registration

Note: Please read the DeepReg Demo Disclaimer.

Source Code

Author

DeepReg Development Team (Shaheer Saeed)

Application

This is a registration between CT images from different patients. The images are all from acquired at the same time-point in the breathing cycle. This is an inter subject registration. This kind of registration is useful for determining how one stimulus affects multiple patients. If a drug or invasive procedure is administered to multiple patients, registering the images from different patients can give medical professionals a sense of how each patient is responding in comparison to others. An example of such an application can be seen in [2].

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- The `demo_data.py`, `demo_train.py` and `demo_predict.py` scripts need to be run using the following command:

```
python3 demos/unpaired_ct_lung/script_name.py
```

A short description of the scripts is provided below. The scripts must be run in the following order:

- Run the `demo_data.py` script: This script does the following:
  - Download data using `tf.keras.utils.get_file`. Data is downloaded to the demo directory but this can be changed (instructions in the comments in the script).
  - Split the data into three sets train, valid and test (change ratio_of_test_and_valid_samples variable to change the ratio of test and valid samples)
  - Restructure the files, for each of the train, valid and test sets, into a directory structure that is suitable for use with the unpaired loader in DeepReg
  - Rescale all images to 0-255 so they are suitable for use with DeepReg
  - Download a pretrained model for use with the predict script
- Run the demo_train.py script: This script does the following:
  - Specify the training options like GPU support
  - Specify the config file paths (the config file to define the network is one which is avialable with DeepReg and the config file for the data is given in the demo folder)
  - Train a network using DeepReg
- Run the demo_predict.py script: This script does the following:
  - Use the pretrained network to make predictions for the test set
  - Use the predictions to plot the results (the path to the images generated in the logs will need to be specified, look at comments in the script to change this)

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/learn2reg_t2_unpaired_train_logs/.

Data

The dataset for this demo comes from [1] and can be downloaded from:

https://zenodo.org/record/3835682#.XsUWXsBpFhE

Tested DeepReg version

Last commit at which demo was tested: c709a46c345552ae1396e6d7ba46a44f7950aea0

Note: This demo was tested using one Nvidia Tesla V100 GPU with a memory of 32GB. Please ensure that enough memory is available to run the demo otherwise memory allocation errors might arise.

Contact

Please raise an issue.

Reference

[1] Hering A, Murphy K, and van Ginneken B. (2020). Lean2Reg Challenge: CT Lung Registration - Training Data [Data set]. Zenodo. http://doi.org/10.5281/zenodo.3835682

[2] Li B, Christensen GE, Hoffman EA, McLennan G, Reinhardt JM. Establishing a normative atlas of the human lung: intersubject warping and registration of volumetric CT images. Acad Radiol. 2003;10(3):255-265. doi:10.1016/s1076-6332(03)80099-5

3.8.3 Unpaired hippocampus MR registration

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

Author

DeepReg Development Team (Adrià Casamitjana)

Application

This is a demo targeting the alignment of hippocampal substructures (head and body) using mono-modal MR images between different patients. The images are cropped around those areas and manually annotated. This is a 3D intra-modal registration using a composite loss of image and label similarity.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- The `demo_data.py`, `demo_train.py` and `demo_predict.py` scripts need to be run using the following command:

```
python3 demos/unpaired_mr_brain/script_name.py
```

A short description of the scripts is provided below. The scripts must be run in the following order:

- (Optional) Create a new configuration file following the template in demos/unpaired_mr_brain/unpaired_mr_brain.yaml. It specifies:
  - Dataset options: input data directory, loader type, data format
  - Model options: backbone network, field type.
  - Training options: losses, optimizer, number of epochs.
- Run the `demo_data.py` script: This script does the following:
  - Download and extract the dataset. Data is downloaded to the demo directory under data/ but this can be changed (instructions in the comments in the script).
  - Split subjects into train/test according to the challenge website.
  - Rescale all images to 0-255 so they are suitable for use with DeepReg
  - Create and apply a binary mask to mask-out the padded values in images.
  - Transform label volumes using one-hot encoding (only for foreground classes)
- Run the `demo_train.py` script: This script does the following:
  - Specify the training options like GPU support
  - Specify the config file paths
  - Train a network using DeepReg
- Run the `demo_predict.py` script: This script does the following:
  - Use the trained network to make predictions for the test set
  - Use the predictions to plot the results (the path to the images generated in the logs will need to be specified, look at comments in the script to change this)

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/learn2reg_t4_unpaired_train_logs/.

Data

The dataset for this demo comes from the Learn2Reg MICCAI Challenge (Task 4) [1] and can be downloaded from:

https://drive.google.com/uc?export=download&id=1RvJIjG2loU8uGkWzUuGjqVcGQW2RzNYA

Tested DeepReg version

Last commit at which demo was tested: v. 0.1.6-alpha

Contact

Please raise an issue.

Reference

[1] AL Simpson et al., *A large annotated medical image dataset for the development and evaluation of segmentation algorithms* (2019). https://arxiv.org/abs/1902.09063

3.8.4 Unpaired prostate ultrasound registration

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

This DeepReg Demo is also an example of cross validation.

Author

DeepReg Development Team

Application

Transrectal ultrasound (TRUS) images are acquired from prostate cancer patients during image-guided procedures. Pairwise registration between these 3D images may be useful for intraoperative motion modelling and group-wise registration for population studies.

Data

The 3D ultrasound images used in this demo were derived from the Prostate-MRI-US-Biopsy dataset, hosted at the Cancer Imaging Archive (TCIA).

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download 10 folds of unpaired 3D ultrasound images;

```
python demos/unpaired_us_prostate_cv/demo_data.py
```

- Call `deepreg_train` from command line. The following example uses three GPUs and launches the first of the ten runs of a 9-fold cross-validation, as specified in the ``*dataset*` section <./unpaired_us_prostate_cv_run1.yaml>`_ and the ``*train*` section <./unpaired_us_prostate_cv_train.yaml>`_, which can be specified in separate yaml files. The 10th fold is reserved for testing;

```
deepreg_train --gpu "1, 2, 3" --config_path demos/unpaired_us_prostate_cv/unpaired_us_
↪prostate_cv_run1.yaml demos/unpaired_us_prostate_cv/unpaired_us_prostate_cv_train.
↪yaml --log_dir unpaired_us_prostate_cv
```

- Call `deepreg_predict` from command line to use the saved ckpt file for testing on the 10th fold data. The following example uses a pre-trained model, on CPU. If not specified, the results will be saved at the created timestamp-named directories under /logs.

```
deepreg_predict --gpu "" --config_path demos/unpaired_us_prostate_cv/unpaired_us_
↪prostate_cv_run1.yaml demos/unpaired_us_prostate_cv/unpaired_us_prostate_cv_train.
↪yaml --ckpt_path demos/unpaired_us_prostate_cv/dataset/pre-trained/weights-
↪epoch5000.ckpt --mode test
```

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/unpaired_us_prostate_cv/.

Tested DeepReg version

Last commit at which demo was tested: 7bec018b5e910f1589888f3f286e9f6a11060c31

Contact

Please raise an issue.

3.9 Grouped Images

The following DeepReg Demos provide examples of using grouped images.

- Pairwise registration for grouped prostate segmentation masks

  This demo registers grouped masks (as input images) of prostate glands from MR images, an example of feature-based registration.

- Pairwise registration for grouped cardiac MR images

  This demo registers grouped CMR images, where each group has multi-sequence CMR images from a single patient.

3.9.1 Pairwise registration for grouped prostate segmentation masks

Note: Please read the DeepReg Demo Disclaimer.

Source Code

This demo uses DeepReg to demonstrate a number of features:

- For grouped data in h5 files, e.g. "group-1-2" indicates the 2th visit from Subject 1;
- Use masks as the images for feature-based registration - aligning the prostate gland segmentation in this case - with deep learning;
- Register intra-patient longitudinal data.

Author

DeepReg Development Team

Application

Longitudinal registration detects the temporal changes and normalises the spatial difference between images acquired at different time-points. For prostate cancer patients under active surveillance programmes, quantifying these changes is useful for detecting and monitoring potential cancerous regions.

Data

This is a demo without real clinical data due to regulatory restrictions. The MR and ultrasound images used are simulated dummy images.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download 10 folds of unpaired 3D ultrasound images and the pre-trained model.

```
python demos/grouped_mask_prostate_longitudinal/demo_data.py
```

- Call `deepreg_train` from command line. The following example uses a single GPU and launches the first of the ten runs of a 9-fold cross-validation, as specified in the ``*dataset*` section <./grouped_mask_prostate_longitudinal_dataset0.yaml>`_ and the ``*train*` section <./grouped_mask_prostate_longitudinal_train.yaml>`_, which can be specified in separate yaml files;

```
deepreg_train --gpu "0" --config_path demos/grouped_mask_prostate_longitudinal/
↪grouped_mask_prostate_longitudinal.yaml --log_dir grouped_mask_prostate_longitudinal
```

- Call `deepreg_predict` from command line to use the saved ckpt file for testing on the data partitions specified in the config file, a copy of which would be saved in the [log_dir]. The following example uses a pre-trained model, on CPU. If not specified, the results will be saved at the created timestamp-named directories under /logs.

```
deepreg_predict --gpu "" --config_path demos/grouped_mask_prostate_longitudinal/
↪grouped_mask_prostate_longitudinal.yaml --ckpt_path demos/grouped_mask_prostate_
↪longitudinal/dataset/pre-trained/weights-epoch500.ckpt --save_png --mode test
```

Pre-trained model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at the dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/grouped_mask_prostate_longitudinal/.

Tested DeepReg version

Last commit at which demo was tested: 3157f880eb99ce10fc3a4a8ebcc595bd67be24e1

Contact

Please raise an issue.

3.9.2 Pairwise registration for grouped cardiac MR images

Note: Please read the DeepReg Demo Disclaimer.

Source Code

This demo uses the grouped dataset loader to register intra-subject multi-sequence cardiac magnetic resonance (CMR) images.

Author

DeepReg Development Team

Application

Computer-assisted management for patients suffering from myocardial infraction (MI) often requires quantifying the difference and comprising the multiple sequences, such as the late gadolinium enhancement (LGE) CMR sequence MI, the T2-weighted CMR. They collectively provide radiological information otherwise unavailable during clinical practice.

Instruction

- Install DeepReg;
- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download all the CMR dataset in a zip file. The script also splits the data into train, val and test sets re-samples all the images to an isotropic voxel size.

```
python demos/grouped_mr_heart/demo_data.py
```

- Call `deepreg_train` from command line. The following example uses a single GPU and launches the first of the ten runs of a 9-fold cross-validation, as specified in the ``*dataset*` section <./grouped_mr_heart_dataset0.yaml>`_ and the ``*train*` section <./grouped_mr_heart_train.yaml>`_, which can be specified in seperate yaml files;

```
deepreg_train --gpu "0" --config_path demos/grouped_mr_heart/grouped_mr_heart.yaml --
↪log_dir grouped_mr_heart
```

- Call `deepreg_predict` from command line to use the saved ckpt file for testing on the data partitions specified in the config file, a copy of which will be saved in the [log_dir]. The following example uses a pre-trained model, on CPU. If not specified, the results will be saved at the created timestamp-named directories under /logs.

```
deepreg_predict --gpu "" --config_path demos/grouped_mr_heart/grouped_mr_heart.yaml --
↪ckpt_path demos/grouped_mr_heart/dataset/pre-trained/weights-epoch500.ckpt --save_
↪png --mode test
```

Pre-trained Model

A pre-trained model will be downloaded after running `demo_data.py` and unzipped at dataset folder under the demo folder. This pre-trained model will be used by default with `deepreg_predict`. Run the user-trained model by specifying with `--ckpt_path` the location where the ckpt files will be saved, in this case (specified by `deepreg_train` as above), /logs/grouped_mr_heart/.

Data

This demo uses CMR images from 45 patients, acquired from the MyoPS2020 challenge held in conjunction with MICCAI 2020.

Tested DeepReg version

Last commit: 74e7b1f749d0df1c140494eba0204f0edd1d7b1e

Contact

Please raise an issue.

Reference

[1] Xiahai Zhuang: Multivariate mixture model for myocardial segmentation combining multi-source images. IEEE Transactions on Pattern Analysis and Machine Intelligence (T PAMI), vol. 41, no. 12, 2933-2946, Dec 2019. link.

[2] Xiahai Zhuang: Multivariate mixture model for cardiac segmentation from multi-sequence MRI. International Conference on Medical Image Computing and Computer-Assisted Intervention, pp.581-588, 2016.

3.10 Classical Registration

The following DeepReg Demos provide examples of using classical registration methods.

- Classical affine registration for head-and-neck CT images

  This demo registers head-and-neck CT images using iterative affine registration.

- Classical nonrigid registration for prostate MR images

  This demo registers prostate MR images using iterative nonrigid registration.

3.10.1 Classical affine registration for head-and-neck CT images

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

This is a special demo that uses the DeepReg package for classical affine image registration, which iteratively solves an optimisation problem. Gradient descent is used to minimise the image dissimilarity function of a given pair of moving anf fixed images.

Author

DeepReg Development Team

Application

Although in this demo the moving images are simulated using a randomly generated transformation. The registration technique can be used in radiotherapy to compensate the difference between CT acquired at different time points, such as pre-treatment and intra-/post-treatment.

Data

https://wiki.cancerimagingarchive.net/display/Public/Head-Neck-PET-CT

Instruction

- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download an example CT volumes with two labels;

```
python demos/classical_ct_headneck_affine/demo_data.py
```

- Run `demo_register.py` script. This script will register two images. The fixed image will be the downloaded data and the moving image will be simulated by applying a random affine transformation, such that the ground-truth is available for. The optimised transformation will be applied to the moving images, as well as the moving labels. The results, saved in a timestamped folder under the project directory, will compare the warped image/labels with the ground-truth image/labels.

```
python demos/classical_ct_headneck_affine/demo_register.py
```

Tested DeepReg version

0.1.0

Contact

Please raise an issue.

Reference

[1] Vallières, M. et al. Radiomics strategies for risk assessment of tumour failure in head-and-neck cancer. Sci Rep 7, 10117 (2017). doi: 10.1038/s41598-017-10371-5

3.10.2 Classical nonrigid registration for prostate MR images

> **Note**: Please read the DeepReg Demo Disclaimer.

Source Code

This is a special demo that uses the DeepReg package for classical nonrigid image registration, which iteratively solves an optimisation problem. Gradient descent is used to minimise the image dissimilarity function of a given pair of moving and fixed images, often regularised by a deformation smoothness function.

Author

DeepReg Development Team

Application

Registering inter-subject prostate MR images may be useful to align different glands in a common space for investigating the spatial distribution of cancer.

Data

https://promise12.grand-challenge.org/

Instruction

- Change current directory to the root directory of DeepReg project;
- Run `demo_data.py` script to download an example MR volumes with the prostate gland segmentation;

```
python demos/classical_mr_prostate_nonrigid/demo_data.py
```

- Run `demo_register.py` script. This script will register two images. The optimised transformation will be applied to the moving images, as well as the moving labels. The results, saved in a timestamped folder under the project directory, will compare the warped image/labels with the ground-truth image/labels.

```
python demos/classical_mr_prostate_nonrigid/demo_register.py
```

Tested DeepReg version

0.1.0

Contact

Please raise an issue.

Reference

[1] Litjens, G., Toth, R., van de Ven, W., Hoeks, C., Kerkstra, S., van Ginneken, B., Vincent, G., Guillard, G., Birbeck, N., Zhang, J. and Strand, R., 2014. Evaluation of prostate segmentation algorithms for MRI: the PROMISE12 challenge. Medical image analysis, 18(2), pp.359-373.

3.11 Command Line Tools

With DeepReg installed, multiple command line tools are available, currently including:

- `deepreg_train`, for training a registration network.
- `deepreg_predict`, for evaluating a trained network.
- `deepreg_warp`, for warping an image with a dense displacement field.

3.11.1 Train

`deepreg_train` accepts the following arguments via command line tools. More configuration can be specified in the configuration file. Please see configuration file for further details.

Required arguments

- **GPU**:

  `--gpu` or `-g`, specifies the index or indices of GPUs for training.

  Example usage:

  - `--gpu ""` for CPU only
  - `--gpu "0"` for using only GPU 0
  - `--gpu "0,1"` for using GPU 0 and 1.

- **Configuration**:

  `--config_path` or `-c`, specifies the configuration file for training.

  The path must end with `.yaml`.

  Optionally, multiple paths can be specified, and the configuration will be merged. In case of conflicts, values are overwritten by the last config file defining them.

  Example usage:

  - `--config_path config1.yaml` for using one single configuration file.
  - `--config_path config1.yaml config2.yaml` for using multiple configuration files.

Optional arguments

- **GPU memory allocation**:

  `--gpu_allow_growth` or `-gr`, if given, TensorFlow will only grow the memory usage as is needed.

  By default it allocates all available GPU memory.

  Example usage:

  - `--gpu_allow_growth`, no extra argument is needed.

- **Load checkpoint**:

  `--ckpt_path` or `-k`, specifies the path of the saved model checkpoint, so that the training will be resumed from the given checkpoint.

  The path must end with `.ckpt`.

By default it starts training from a random initialization.

Example usage:

- `--ckpt_path weights-epoch2.ckpt` for reloading the given checkpoint.

- **Output root**:

`--log_root`, specifies the directory for saving all logs.

By default it is `logs` under the root of package.

Example usage:

- `--log_root /logs` for saving all logs under `/logs`.

- **Output directory**:

`--log_dir` or `-l`, specifies the directory name to save logs.

The directory will be under the `log_root` which is `logs` by default.

By default it creates a timestamp-named directory, e.g. `logs/20200810-194042/`.

Example usage:

- `--log_dir test` for saving under `logs/test/`.

Output

During the training, multiple output files will be saved in the log directory `logs/log_dir`, where `log_dir` is specified in the arguments, otherwise a timestamped folder name will be used. The output files are:

- `config.yaml` is a backup of the used configuration. It can be used for prediction. In case of multiple configuration files, a merged configuration file will be saved.
- `train/` and `validation/` are the directories that save tensorboard logs on metrics.
- `save/` is the directory containing saved checkpoints of the trained network.

3.11.2 Predict

`deepreg_predict` accepts the following arguments via command line tools. More configuration can be specified in the configuration file. Please see configuration file for further details.

Required arguments

- **GPU**:

`--gpu` or `-g`, specifies the index or indices of GPUs for training.

Example usage:

- `--gpu ""` for CPU only
- `--gpu "0"` for using only GPU 0
- `--gpu "0,1"` for using GPU 0 and 1.

- **Model checkpoint**:

  `--ckpt_path` or `-k`, specifies the path of the saved model checkpoint, so that the trained model will be loaded for evaluation.

  The path must end with `.ckpt`.

  Example usage:

  - `--ckpt_path weights-epoch2.ckpt` for reloading the given checkpoint.

- **Evaluation data**:

  `--mode` or `-m`, specifies in which data set the prediction is performed.

  It must be one of `train` / `valid` / `test`.

  Example usage:

  - `--mode test` for evaluating the model on test data.

Optional arguments

- **GPU memory allocation**:

  `--gpu_allow_growth` or `-gr`, if given, TensorFlow will only grow the memory usage as is needed.

  By default it allocates all availables in the GPU memory.

  Example usage:

  - `--gpu_allow_growth`, no extra argument is needed.

- **Output root**:

  `--log_root`, specifies the directory for saving all logs.

  By default it is `logs` under the root of package.

  Example usage:

  - `--log_root /logs` for saving all logs under `/logs`.

- **Output directory**:

  `--log_dir` or `-l`, specifies the directory name to save logs.

  The directory will be under the `log_root` which is `logs` by default.

  By default it creates a timestamp-named directory, e.g. `logs/20200810-194042/`.

  Example usage:

  - `--log_dir test` for saving under `logs/test/`.

- **Batch size**:

  `--batch_size` or `-b`, specifies the mini-batch size (per GPU) for prediction.

  The default value is 1.

  Example usage:

  - `--batch_size 2` for using a mini-batch size of 2.

- **Save outputs in Nifti format**:

  The predicted 3D tensors can be saved in Nifti format for further calculation.

  By default it saves outputs in Nifti format.

  Example usage:

  - `--save_nifti`, for saving the outputs in Nifti format.
  - `--no_nifti`, for not saving the outputs in Nifti format.

- **Save outputs in png format**:

  The predicted 3D tensors can be saved as a slice of 2D images for quick visualization.

  As values have to be normalized between 0~255 (or 0~1) for png files (Nifti files are not impacted), all images (`moving_image`, `fixed_image` and `pred_fixed_image`) and displacement/velocity fields (`ddf` and `dvf`) will be normalized before being saved. Labels (`moving_label`, `fixed_label` and `pred_fixed_label`) are not affected as they are already within 0~1.

  By default it saves the outputs in png format.

  Example usage:

  - `--save_png`, for saving the outputs in png format.
  - `--save_png`, for not saving the outputs in png format.

- **Configuration**:

  `--config_path` or `-c`, specifies the configuration file for prediction.

  The path must end with `.yaml`.

  By default it uses the configuration file saved in the directory of the given checkpoint.

  Example usage:

  - `--config_path config1.yaml` for using one single configuration file.

Output

During the evaluation, multiple output files will be saved in the log directory `logs/log_dir/mode` where

- `log_dir` is defined in arguments, or a timestamped folder name will be used;
- `mode` is `train` or `valid` or `test`, specified by the argument.

The saved files include:

- Metrics to evaluate the registration performance
  - `metrics.csv` saves the metrics on all samples. Each line corresponds to a data sample.
  - `metrics_stats_per_label.csv` saves the mean, median and std of each metrics on all samples with the same label index.
  - `metrics_stats_overall.csv` saves a set of commonly used statistics (such as mean and std) on the metrics over all samples.
- Inputs and predictions for each pair of image.

  Each pair has its own directory and the followings tensors are saved inside if available. Tensors can be saved in Nifti format (one single file) or in png format (one folder contains all image slices, ordered by depth) or both.

  - `ddf`, `dvf`, `affine`

    DDF stands for dense displacement field; DVF stands for dense (static) velocity field.

    The 12 parameters of affine transformation are saved in `affine.txt`.

  - `moving_image`, `fixed_image` and `pred_fixed_image`

    `pred_fixed_image` is the warped moving image if the network predicts a DDF or a DVF or an affine transformation.

  - `moving_label`, `fixed_label` and `pred_fixed_label` under directory `label_i` if the sample is labeled and `i` is the label index.

    `pred_fixed_label` is the predicted label in the fixed image space. In many cases, this is equivalent to the warped moving label, if the network predicts a DDF or a DVF or an affine transformation.

3.11.3 Warp

`deepreg_warp` accepts the following arguments:

Required arguments

- **Image file**:

  `--image` or `-i`, specifies the file path of the image/label.

  The image/label should be saved in a Nifti file with suffix `.nii` or `.nii.gz`. The image/label should be a 3D / 4D tensor, where the first three dimensions correspond to the moving image shape and the fourth can be a channel of features.

  Example usage:

  - `--image input_image.nii.gz`

- **DDF file**:

  `--ddf` or `-d`, specifies the file path of the DDF.

  The DDF should be saved in a Nifti file with suffix `.nii` or `.nii.gz`. The DDF should be a 4D tensor, where the first three dimensions correspond to the fixed image shape and the fourth dimension has 3 channels corresponding to x, y, z axes.

  Example usage:

  - `--image input_DDF.nii.gz`

Optional arguments

- **Output directory**:

  `--out` or `-o`, specifies the file path for the output.

  The path should end with `.nii` or `.nii.gz`, otherwise the output path will be corrected automatically based on the given path.

  By default it saves the output as `warped.nii.gz` in the current directory.

  Example usage:

  - `--out output_image.nii.gz`

Output

The warped image is saved in the given output file path, otherwise the default file path `warped.nii.gz` will be used.

3.11.4 Visualise

In addition to the images in the output, DeepReg provides a set of tools with the command `deepreg_vis`. See more details in its usage documentation.

3.12 Configuration File

Besides the arguments provided to the command line tools, detailed training and prediction configuration is specified in a `yaml` file. The configuration file contains two sections, `dataset` and `train`.

3.12.1 Dataset section

See the dataset loader configuration for more details.

3.12.2 Train section

The `train` section defines the neural network training hyper-parameters, by specifying subsections, `model`, `loss`, `optimizer`, `preprocess` and other training hyper-parameter, including `epochs` and `save_period`. See an example configuration, with comments on the available options in each subsection.

This section is highly application-specific. More examples can be found in DeepReg Demos.

3.13 Dataset Loader

3.13.1 Dataset type

DeepReg provides six dataset loaders to support the following three different types of datasets:

- **Paired images**

  Images are organized into moving and fixed image pairs.

  An example case is two-modalities intra-subject registration, such as registering one subject's MR image to the corresponding ultrasound image.

- **Unpaired images**

  Images may be considered independent samples.

  An example case is single-modality inter-subject registration, such as registering one CT image to another from different subjects.

- **Grouped images**

  Images are organized into multiple groups.

  An example case is single-modality intra-subject registration, such as registering time-series images within individual subjects, a group is one subject in this case.

For all three above cases, the images can be either unlabeled or labeled. A label is represented by a boolean mask on the image, such as a segmentation of an anatomical structure or landmark.

3.13.2 Dataset requirements

To use the provided dataset loaders, other detailed images and labels requirements are described in individual dataset loader sections. General requirements are described as follows.

- Image
  - DeepReg currently supports 3D images. But images do not have to be of the same shape, and it will be resized to the required shape using linear interpolation.
  - Currently, DeepReg only supports images stored in Nifti files or H5 files. Check Nifti_loader and h5_loader for more details.
  - **Images are automatically normalized** at per-image level: the intensity values x equals to `(x-min(x)+EPS) / (max(x)-min(x)+EPS)` so that its values are between [0,1]. Check `GeneratorDataLoader.data_generator` in loader interface for more details.
- Label
  - If an image is labeled, the label shape is recommended to be the same as the image shape. Otherwise, the resize might give unexpected behaviours. But each image can have more than one labels.

    For instance, an image of shape `(dim1, dim2, dim3)`, its label shape can be `(dim1, dim2, dim3)` (single label) or `(dim1, dim2, dim3, num_labels)` (multiple labels).
  - **All labels are assumed to have values between [0, 1].** So DeepReg accepts binary segmentation masks or soft labels with float values between [0,1]. This is to prevent accidental use of non-one-hot encoding to represent multiple class labels. In case of multi labels, please use one-hot encoding to transform them into multiple channels such that each class has its own binary label.
  - When the images are paired, the moving and fixed images must have the same number of labels.
  - When there are multiple labels, it is assumed that the labels are ordered, such that the channel of index `label_idx` is the same anatomical or pathological structure.
  - Currently, if the data are labeled, each data sample must have at least one label. For missing labels, consider using all zeros as a workaround.

3.13.3 Paired images

For paired images, each pair contains a moving image and a fixed image. Optionally, corresponding moving label(s) and fixed label(s).

Specifically, given a pair of images

- When the image is unlabeled,
  - moving image of shape `(m_dim1, m_dim2, m_dim3)`
  - fixed image of shape `(f_dim1, f_dim2, f_dim3)`
- When the image is labeled and there is only one label,
  - moving image of shape `(m_dim1, m_dim2, m_dim3)`
  - fixed image of shape `(f_dim1, f_dim2, f_dim3)`
  - moving label of shape `(m_dim1, m_dim2, m_dim3)`

  - fixed label of shape `(f_dim1, f_dim2, f_dim3)`
- When the image is labeled and there are multiple labels,
  - moving image of shape `(m_dim1, m_dim2, m_dim3)`
  - fixed image of shape `(f_dim1, f_dim2, f_dim3)`
  - moving label of shape `(m_dim1, m_dim2, m_dim3, num_labels)`
  - fixed label of shape `(f_dim1, f_dim2, f_dim3, num_labels)`

Sampling

For paired images, one epoch of the dataset iterates all the image pairs sequentially with random orders. So each image pair is sampled once in each epoch with equal chance. For validation or testing, the random seed is fixed to ensure consistency.

When an image has multiple labels, e.g. the segmentation of different organs in a CT image, only one label will be sampled during training. In particular, only corresponding labels will be sampled between a pair of moving and fixed images. In case of validation or testing, instead of sampling one label per image, all labels will be iterated.

Configuration

An example configuration for paired dataset is provided as follows.

```yaml
dataset:
  dir:
    train: "data/test/h5/paired/train" # folder contains training data
    valid: "data/test/h5/paired/test" # folder contains validation data
    test: "data/test/h5/paired/test" # folder contains test data
  format: "nifti" # value should be nifti / h5
  type: "paired" # value should be paired / unpaired / grouped
  labeled: true # value should be true / false
  moving_image_shape: [16, 16, 16] # value should be like [dim1, dim2, dim3]
  fixed_image_shape: [8, 8, 8] # value should be like [dim1, dim2, dim3]
```

where, the configuration can be split into common configurations that shared by all dataset types and specific configurations for paired images:

- Common configurations
  - `dir/train` gives the directory containing training data. Same for `dir/valid` and `dir/test`.
  - `format` can only be Nifti or h5 currently.
  - `type` can be paired, unpaired or grouped, corresponding to the dataset type described above.
  - `labeled` is a boolean indicating if the data is labeled or not.
- Paired images configurations
  - `moving_image_shape` is the shape of moving images, a list of three integers.
  - `fixed_image_shape` is the shape of fixed images, a list of three integers.

Optionally, multiple dataset directories can be specified, such that the data will be sampled from several directories, for instance:

```yaml
dataset:
  dir:
    train: # folder contains training data
      - "data/test/h5/paired/train1"
      - "data/test/h5/paired/train2"
    valid: "data/test/h5/paired/test" # folder contains validation data
    test: "data/test/h5/paired/test" # folder contains test data
```

This is particularly useful when performing an experiment such as cross-validation.

File loader

For paired data, the specific requirements for data stored in Nifti and h5 files are described as follows.

Nifti

Nifti data are stored in files with suffix `.nii.gz`. Each file should contain only one 3D or 4D tensor, corresponding to an image or a label.

`obs` is short for one observation of a data sample - a 3D image volume or a 3D/4D label volume - and the name can be any string.

All image data should be placed under `moving_images/`, `fixed_images/` with respect to the provided directory. The label data should be placed under `moving_labels/`, and `fixed_labels/`, if available. These are *top* directories.

File names should be consistent between top directories, e.g.:

- moving_images/
  - obs1.nii.gz
  - obs2.nii.gz
  - ...
- fixed_images/
  - obs1.nii.gz
  - obs2.nii.gz
  - ...
- moving_labels/
  - obs1.nii.gz
  - obs2.nii.gz
  - ...
- fixed_labels/
  - obs1.nii.gz
  - obs2.nii.gz
  - ...

Check test paired Nifti data as an example.

Optionally, the data may not be all saved directly under the top directory. They can be further grouped in subdirectories as long as the data paths are consistent.

H5

H5 data are stored in files with suffix `.h5`. Hierarchical multi-level indexing is not used. Each file should contain multiple key-value pairs and values are 3D or 4D tensors. Each file is equivalent to a top folder in Nifti cases.

All image data should be stored in `moving_images.h5`, `fixed_images.h5`. The label data should be stored in `moving_labels.h5`, and `fixed_labels.h5`, if available.

The keys should be consistent between files, e.g.:

- moving_images.h5 has keys:
  - "obs1"
  - "obs2"
  - ...
- fixed_images.h5 has keys:
  - "obs1"
  - "obs2"
  - ...
- moving_labels.h5 has keys:
  - "obs1"
  - "obs2"
  - ...
- fixed_labels.h5 has keys:
  - "obs1"
  - "obs2"
  - ...

Check test paired H5 data as an example.

3.13.4 Unpaired images

For unpaired images, all images are considered as independent and they must have the same shape. Optionally, there are corresponding labels for the images.

Specifically,

- When the image is unlabeled,
  - image of shape `(dim1, dim2, dim3)`
- When the image is labeled and there is only one label,
  - image of shape `(dim1, dim2, dim3)`
  - label of shape `(dim1, dim2, dim3)`

- When the image is labeled and there are multiple labels,
  - image of shape `(dim1, dim2, dim3)`
  - label of shape `(dim1, dim2, dim3, num_labels)`

Sampling

During each epoch, image pairs will be sampled without replacement. Therefore, given N images, one epoch will thereby have floor(N / 2) image pairs. For validation or testing, the random seed is fixed to ensure consistency.

In case of multiple labels, the sampling method is the same as in *paired data*. In particular, the only corresponding label pairs will be sampled between the two sampled images.

Configuration

An example configuration for unpaired dataset is provided as follows.

```
dataset:
  dir:
    train: "data/test/h5/paired/train" # folder contains training data
    valid: "data/test/h5/paired/test" # folder contains validation data
    test: "data/test/h5/paired/test" # folder contains test data
  format: "nifti" # value should be nifti / h5
  type: "unpaired" # value should be paired / unpaired / grouped
  labeled: true # value should be true / false
  image_shape: [16, 16, 16] # value should be like [dim1, dim2, dim3]
```

where

- Common configurations

  Same as *paired images*.

- Unpaired images configurations
  - `image_shape` is the shape of images, a list of three integers.

File loader

For unpaired data, the specific requirements for data stored in nifti and h5 files are described as follows.

Nifti

Nifti data are stored in files with suffix `.nii.gz` or `.nii`. Each file must contain only one 3D or 4D tensor, corresponding to an image or a label.

`obs` is short for one observation of a data sample - a 3D image volume or a 3D/4D label volume - and the name can be any string.

All image data should be placed under `images/`. The label data should be placed under `labels/`, if available. These are *top* directories.

File names should be consistent between top directories, e.g.:

- images/
  - obs1.nii.gz

  - obs2.nii.gz
  - ...
- labels/
  - obs1.nii.gz
  - obs2.nii.gz
  - ...

Check test unpaired Nifti data as an example.

H5

F5 data are stored in files with suffix `.h5`. Hierarchical multi-level indexing is not used. Each file should contain multiple key-value pairs and values are 3D or 4D tensors. Each file is equivalent to a top folder in Nifti cases.

All image data should be placed under `images.h5`. The label data should be placed under `labels.h5`, if available.

The keys should be consistent between files, e.g.:

- images.h5 has keys:
  - "obs1"
  - "obs2"
  - ...
- labels.h5 has keys:
  - "obs1"
  - "obs2"
  - ...

Check test unpaired H5 data as an example.

3.13.5 Grouped images

For grouped images, images may not be paired but organized into multiple groups. Each group must have at least two images.

The requirements are the same as unpaired images. Specifically,

- When the image is unlabeled,
  - image of shape `(dim1, dim2, dim3)`
- When the image is labeled and there is only one label,
  - image of shape `(dim1, dim2, dim3)`
  - label of shape `(dim1, dim2, dim3)`
- When the image is labeled and there are multiple labels,
  - image of shape `(dim1, dim2, dim3)`
  - label of shape `(dim1, dim2, dim3, num_labels)`

Sampling

For sampling image pairs, DeepReg provides the following options:

- **inter-group sampling**, where the moving image and fixed image come from different groups.
- **intra-group sampling**, where the moving image and fixed image come from the same group.
- **mixed sampling**, where the image pairs are mixed from inter-group sampling and intra-group sampling.

For validation or testing, the random seed is fixed to ensure consistency.

In case of multiple labels, the sampling method is the same as *paired data*. In particular, only the corresponding label pairs will be sampled between the two sampled images.

Intra-group

To form image pairs, the group and image are sampled sequentially at two stages,

1. Sample a group from which the moving and fixed images will be sampled.
2. Sample two different images from the group as moving and fixed images. When sampling images from the same group, there are multiple options, denoted by `intra_group_option`:
    - `forward`: the moving image always has a smaller image index than fixed image.
    - `backward`: the moving image always has a larger image index than fixed image.
    - `unconstrained`: no constraint on the image index as long as the two images are different.

Therefore, each epoch generates the same number of image pairs as the number of groups, where all groups will be first shuffled and iterated. The `intra_group_option` is useful in implementing temporal-order sensitive sampling strategy.

Inter-group

To form image pairs, the group and image are sampled sequentially at two stages,

1. Sample the first group, from which the moving image will be sampled.
2. Sample the second group, from which the fixed image will be sampled.
3. Sample an image from the first group as moving image.
4. Sample an image from the second group as fixed image.

Therefore, each epoch generates the same number of image pairs as the number of groups, where all groups will be first shuffled and iterated.

Mixed

Optionally, it is possible to mix inter-group and intra-group sampling by specifying the intra-group image sampling probability `intra_group_prob`=[0,1]. The value 0 means entirely inter-group sampling and 1 means entirely intra-group sampling.

Given 0<p<1, when generating intra-group pairs, there is (1-p)*100% chance to sample the fixed images from a different group, after sampling the moving image from the current intra-group images.

Iterated

Optionally, it is possible to generate all combinations of inter-/intra-group image pairs, with `sample_image_in_group` set to false. This is originally designed for evaluation. Mixing inter-/intra-group sampling is not supported with with `sample_image_in_group` set to false.

Configuration

An example configuration for grouped dataset is provided as follows.

```yaml
dataset:
  dir:
    train: "data/test/h5/paired/train" # folder contains training data
    valid: "data/test/h5/paired/test" # folder contains validation data
    test: "data/test/h5/paired/test" # folder contains test data
  format: "nifti" # value should be nifti / h5
  type: "unpaired" # value should be paired / unpaired / grouped
  labeled: true # value should be true / false
  intra_group_prob: 1 # probability of intra-group sampling, value should be between␣
↪0 and 1
  intra_group_option: "forward" # option for intra-group sampling, value should be␣
↪forward / backward / unconstrained
  sample_image_in_group: true # true if sampling one image per group, value should be␣
↪true / false
  image_shape: [16, 16, 16] # value should be like [dim1, dim2, dim3]
```

where

- Common configurations

  Same as *paired images*.

- Grouped images configurations
  - `intra_group_prob`, a value between 0 and 1, 0 is for inter-group only and 1 is for intra-group only.
  - `intra_group_option`, forward or backward or unconstrained, as described above.
  - `sample_image_in_group`, true if sampling one image at a time per group, false if generating all possible pairs.

File loader

For grouped data, the specific requirements for data stored in Nifti and h5 files are described as follows.

Nifti

Nifti data are stored in files with suffix `.nii.gz`. Each file should contain only one 3D or 4D tensor, corresponding to an image or a label.

`obs` is short for one observation of a data sample - a 3D image volume or a 3D/4D label volume - and the name can be any string.

All image data should be placed under `images/`. The label data should be placed under `labels/`, if available. These are *top* directories.

The leaf directories will be considered as different groups, and file names should be consistent between top directories, e.g.:

- images
  - group1
    - obs1.nii.gz
    - obs2.nii.gz
    - ...
  - ...
- labels
  - group1
    - obs1.nii.gz
    - obs2.nii.gz
    - ...
  - ...

Check test grouped Nifti data as an example.

H5

H5 data are stored in files with suffix `.h5`. Hierarchical multi-level indexing is not used. Each file should contain multiple key-value pairs and values are 3D or 4D tensors. Each file is equivalent to a top folder in Nifti cases.

All image data should be placed under `images.h5`. The label data should be placed under `labels.h5`, if available.

The keys must satisfy a specific format, `group-%d-%d`, where `%d` represents an integer number. The first number corresponds to the group index, and the second number corresponds to the observation index. For example, `group-3-2` corresponds to the second observation from the third group.

The keys should be consistent between files, e.g.:

- images.h5 has keys:
  - "group-1-1"
  - "group-1-2"
  - ...
  - "group-2-1"
  - ...
- labels.h5 has keys:
  - "group-1-1"
  - "group-1-2"
  - ...
  - "group-2-1"
  - ...

Check test grouped H5 data as an example.

3.14 Experimental Features

DeepReg provides some experimental features. These are still in development with variable levels of readiness.

The following tutorials provide an overview of these features. To submit feedback, open a new issue.

- Label sampling

3.14.1 Label sampling

Images may have multiple labels, such as with segmentation of different organs in CT scans. In this case, for each sampled image pair, one label pair is randomly chosen by default.

Corresponding label pairs

When using multiple labels, ensure the labels are ordered correctly. `label_idx` in `[width, height, depth, label_idx]` must be the same anatomical or pathological structure; a corresponding label pair between the moving and fixed labels.

Consistent label pairs

Consistent label pairs between a pair of moving and fixed labels requires:

1. The two images have the same number of labels, and
2. The labels have the same order

When a pair of moving and fixed images have inconsistent label pairs, label dissimilarity cannot be defined. The following applies:

- When using the unpaired-labeled-image loader, consistent label pairs are required;
- When using the grouped-labeled-image loader, consistent label pairs are required between intra-group image pairs;
- When mixing intra-inter-group images in the grouped-labeled-image loader, consistent label pairs are required between all intra-group and inter-group image pairs.

However,

- When using the paired-labeled-image loader, consistent label pairs are not required between different image pairs;
- When using the grouped-labeled-image loader without mixing intra-group and inter-group images, consistent label pairs are not required between different image groups.

Partially labeled image data

When one of the label dissimilarity measures prevents accidentally missing labels. When appropriate, enable training with missing labels with a placeholder all-zero mask for the imaging data.

Option for iterating all available label pairs

This option is default for testing. All the label pairs will be sampled once for each sampled image pair. This option is not supported when mixing intra-group and inter-group image pairs.

3.15 Entry Point

3.15.1 Train

Module to train a network using init files and a CLI

deepreg.train.**build_callbacks**(*log_dir: str*, *histogram_freq: int*, *save_period: int*) → list
Function to prepare callbacks for training.

Parameters

- **log_dir** – directory of logs
- **histogram_freq** – save the histogram every X epochs
- **save_period** – save the checkpoint every X epochs

Returns a list of callbacks

deepreg.train.**build_config**(*config_path: (<class 'str'>, <class 'list'>)*, *log_root: str*, *log_dir: str*, *ckpt_path: str*) → [<class 'dict'>, <class 'str'>]
Function to initialise log directories, assert that checkpointed model is the right type and to parse the configuration for training

Parameters

- **config_path** – list of str, path to config file
- **log_root** – str, root of logs
- **log_dir** – str, path to where training logs to be stored.
- **ckpt_path** – str, path where model is stored.

Returns

- config: a dictionary saving configuration
- log_dir: the path of directory to save logs

deepreg.train.**main**(*args=None*)
Entry point for train script

deepreg.train.**train**(*gpu: str*, *config_path: (<class 'str'>, <class 'list'>)*, *gpu_allow_growth: bool*, *ckpt_path: str*, *log_dir: str*, *log_root: str = 'logs'*)
Function to train a model

Parameters

- **gpu** – str, which local gpu to use to train
- **config_path** – str, path to configuration set up
- **gpu_allow_growth** – bool, whether or not to allocate whole GPU memory to training
- **ckpt_path** – str, where to store training checkpoints
- **log_root** – str, root of logs

- **log_dir** – str, where to store logs in training

3.15.2 Predict

Module to perform predictions on data using command line interface

deepreg.predict.**build_config**(*config_path: (<class 'str'>, <class 'list'>)*, *log_root: str*, *log_dir: str*, *ckpt_path: str*) → [<class 'dict'>, <class 'str'>]

Function to create new directory to log directory to store results.

Parameters

- **config_path** – string or list of strings, path of configuration files
- **log_root** – str, root of logs
- **log_dir** – string, path to store logs.
- **ckpt_path** – str, path where model is stored.

Returns

- config, configuration dictionary
- log_dir, path of the directory for saving outputs

deepreg.predict.**build_pair_output_path**(*indices: list*, *save_dir: str) -> (<class 'str'>, <class 'str'>*)

Create directory for saving the paired data

Parameters

- **indices** – indices of the pair, the last one is for label
- **save_dir** – directory of output

Returns

- save_dir, str, directory for saving the moving/fixed image
- label_dir, str, directory for saving the rest outputs

deepreg.predict.**main**(*args=None*)

Function to run in command line with argparse to predict results on data for a given model

deepreg.predict.**predict**(*gpu: str*, *gpu_allow_growth: bool*, *ckpt_path: str*, *mode: str*, *batch_size: int*, *log_dir: str*, *sample_label: str*, *config_path: (<class 'str'>, <class 'list'>)*, *save_nifti: bool = True*, *save_png: bool = True*, *log_root: str = 'logs'*)

Function to predict some metrics from the saved model and logging results.

Parameters

- **gpu** – str, which env gpu to use.
- **gpu_allow_growth** – bool, whether to allow gpu growth or not
- **ckpt_path** – str, where model is stored, should be like log_folder/save/xxx.ckpt
- **mode** – train / valid / test, to define which split of dataset to be evaluated
- **batch_size** – int, batch size to perform predictions in
- **log_dir** – str, path to store logs
- **sample_label** – sample/all, not used

- **save_nifti** – if true, outputs will be saved in nifti format
- **save_png** – if true, outputs will be saved in png format
- **config_path** – to overwrite the default config

deepreg.predict.**predict_on_dataset**(*dataset: tensorflow.data.Dataset*, *fixed_grid_ref: tensorflow.Tensor*, *model: tensorflow.keras.Model*, *model_method: str*, *save_dir: str*, *save_nifti: bool*, *save_png: bool*)

Function to predict results from a dataset from some model

Parameters

- **dataset** – where data is stored
- **fixed_grid_ref** – shape=(1, f_dim1, f_dim2, f_dim3, 3)
- **model** – model to be used for prediction
- **model_method** – str, ddf / dvf / affine / conditional
- **save_dir** – str, path to store dir
- **save_nifti** – if true, outputs will be saved in nifti format
- **save_png** – if true, outputs will be saved in png format

3.15.3 Warp

Module to warp a image with given ddf. A CLI tool is provided.

deepreg.warp.**main**(*args=None*)

Entry point for warp script

deepreg.warp.**warp**(*image_path: str*, *ddf_path: str*, *out_path: str*)

Parameters

- **image_path** – file path of the image file
- **ddf_path** – file path of the ddf file
- **out_path** – file path of the output

3.16 Dataset Loader

3.16.1 Paired Loader

Loads paired image data supports h5 and Nifti formats supports labeled and unlabeled data

class deepreg.dataset.loader.paired_loader.**PairedDataLoader**(*file_loader, data_dir_paths: List[str], labeled: bool, sample_label: str, seed, moving_image_shape: (<class 'list'>, <class 'tuple'>), fixed_image_shape: (<class 'list'>, <class 'tuple'>)*)

Loads paired data using given file loader Handles both labeled and unlabeled cases The function sample_index_generator needs to be defined for the GeneratorDataLoader class

Parameters

- **file_loader** –
- **data_dir_paths** – path of the directories storing data, the data has to be saved under four different sub-directories: moving_images, fixed_images, moving_labels, fixed_labels
- **labeled** – true if the data are labeled
- **sample_label** –
- **seed** –
- **moving_image_shape** – (width, height, depth)
- **fixed_image_shape** – (width, height, depth)

sample_index_generator()

Generate indexes in order to load data using the GeneratorDataLoader class

validate_data_files()

Verify all loaders have the same files

3.16.2 Unpaired Loader

Loads unpaired data supports h5 and Nifti formats supports labeled and unlabeled data

class deepreg.dataset.loader.unpaired_loader.**UnpairedDataLoader**(*file_loader, data_dir_paths: List[str], labeled: bool, sample_label: str, seed, image_shape: (<class 'list'>, <class 'tuple'>)*)

Loads unpaired data using given file loader, handles both labeled and unlabeled cases The function sample_index_generator needs to be defined for the GeneratorDataLoader class

Load data which are unpaired, labeled or unlabeled

Parameters

- **file_loader** –

- **data_dir_paths** – paths of the directories storing data, the data has to be saved under four different sub-directories: images, labels
- **sample_label** –
- **seed** –
- **image_shape** – (width, height, depth)

close()
Close the moving files opened by the file_loaders

sample_index_generator()
Generates sample indexes in order to load data using the GeneratorDataLoader class

validate_data_files()
Verify all loader have the same files. Since fixed and moving loaders come from the same file_loader, there's no need to check both (avoid duplicate)

3.16.3 Grouped Loader

Loads grouped data supports h5 and Nifti formats supports labeled and unlabeled data Read https://deepreg.readthedocs.io/en/latest/api/loader.html#module-deepreg.dataset.loader.grouped_loader for more details.

class deepreg.dataset.loader.grouped_loader.**GroupedDataLoader**(*file_loader, data_dir_paths: List[str], labeled: bool, sample_label: (<class 'str'>, None), intra_group_prob: float, intra_group_option: str, sample_image_in_group: bool, seed: (<class 'int'>, None), image_shape: (<class 'list'>, <class 'tuple'>)*)
Loads grouped data sample_index_generator from GeneratorDataLoader is defined to yield indexes of images to load AbstractUnpairedLoader handles different file formats

Parameters

- **file_loader** – a subclass of FileLoader
- **data_dir_paths** – paths of the directory storing data, the data has to be saved under two different sub-directories:
  - images
  - labels
- **labeled** – bool, true if the data is labeled, false if unlabeled
- **sample_label** – "sample" or "all", read *get_label_indices* in deepreg/dataset/util.py for more details.

- **intra_group_prob** – float between 0 and 1,
  - 0 means generating only inter-group samples,
  - 1 means generating only intra-group samples
- **intra_group_option** – str, "forward", "backward, or "unconstrained"
- **sample_image_in_group** – bool,
  - if true, only one image pair will be yielded for each group, so one epoch has num_groups pairs of data,
  - if false, iterate through this loader will generate all possible pairs
- **seed** – controls the randomness in sampling, if seed=None, then the randomness is not fixed
- **image_shape** – list or tuple of length 3, corresponding to (dim1, dim2, dim3) of the 3D image

close()
Close file loaders

get_inter_sample_indices() → list
Calculate the sample indices for inter-group sampling The index to identify a sample is (group1, image1, group2, image2), means

- image1 of group1 is moving image
- image2 of group2 is fixed image

All pairs of images in the dataset are registered. Assuming group i has ni images, and that N=[n1, n2, . . . , nI], then in total the number of samples are: sum(N) * (sum(N)-1) - sum(N * (N-1))

Returns a list of sample indices

get_intra_sample_indices() → list
Calculate the sample indices for intra-group sampling The index to identify a sample is (group1, image1, group2, image2), means - image1 of group1 is moving image - image2 of group2 is fixed image

Assuming group i has ni images, then in total the number of samples are - sum(ni * (ni-1) / 2) for forward/backward - sum(ni * (ni-1)) for unconstrained

Returns a list of sample indices

sample_index_generator()
Yield (moving_index, fixed_index, image_indices) sequentially, where

- moving_index = (group1, image1)
- fixed_index = (group2, image2)
- image_indices = [group1, image1, group2, image2]

validate_data_files()
If the data are labeled, verify image loader and label loader have the same files

3.17 File Loader

3.17.1 Interface

class deepreg.dataset.loader.interface.**FileLoader**(*dir_paths: list*, *name: str*, *grouped: bool*)

Interface / abstract class to load data from multiple directories

Parameters

- **dir_paths** – path to the directory of the data set
- **name** – name is used to identify the subdirectories or file names
- **grouped** – true if the data is grouped

close()

Close opened file handles if exist.

get_data(*index: (<class 'int'>, <class 'tuple'>)*)

Get one data array by specifying an index.

Parameters **index** – the data index which is required

- for paired or unpaired, the index is one single int, data_index
- for grouped, the index is a tuple of two ints, (group_index, in_group_data_index)

Returns the data array at the specified index

get_data_ids()

Return the unique IDs of the data in this data set. This function is used to verify the consistency between moving and fixed images and label.

get_num_groups() → int

Return the number of groups in grouped data set.

Returns int, number of groups in this data set, if grouped

get_num_images() → int

Return the number of image in this data set.

Returns int, number of images in this data set

get_num_images_per_group() → List[int]

Return the number of images in each group. Each group must have at least one image.

Returns a list of integers, representing the number of images in each group.

set_data_structure()

Store the data structure in the memory so that we can retrieve data using data_index

set_group_structure()

In addition to set_data_structure, store the group structure in the group_struct so that group_struct[group_index] = list of data_index and data can be retrieved data by data_index = group_struct[group_index][in_group_data_index]

3.17.2 Nifti Loader

class deepreg.dataset.loader.nifti_loader.**NiftiFileLoader**(*dir_paths: List[str], name: str, grouped: bool*)

Generalized loader for nifti files

Parameters

- **dir_paths** – path to the directory of the data set
- **name** – name is used to identify the subdirectories or file names
- **grouped** – true if the data is grouped

close()
Close opened files

get_data(*index: (<class 'int'>, <class 'tuple'>)*) → numpy.ndarray
Get one data array by specifying an index

Parameters **index** – the data index which is required

- for paired or unpaired, the index is one single int, data_index
- for grouped, the index is a tuple of two ints, (group_index, in_group_data_index)

Returns arr the data array at the specified index

get_data_ids()
Return the unique IDs of the data in this data set this function is used to verify the consistency between images and label, moving and fixed

Returns data_path_splits but without suffix

get_num_images() → int

Returns int, number of images in this data set

set_data_structure()
Store the data structure in the memory so that we can retrieve data using data_index this function sets data_path_splits, a list of string tuples to identify path of data

- if grouped, a split is (dir_path, group_path, file_name, suffix) data is stored in dir_path/name/group_path/file_name.suffix
- if not grouped, a split is (dir_path, file_name, suffix) data is stored in dir_path/name/file_name.suffix

set_group_structure()
In addition to set_data_structure store the group structure in the group_struct so that group_struct[group_index] = list of data_index we can retrieve data using (group_index, in_group_data_index) data_index = group_struct[group_index][in_group_data_index]

deepreg.dataset.loader.nifti_loader.**load_nifti_file**(*file_path: str*) → numpy.ndarray

Parameters **file_path** – path of a Nifti file with suffix .nii or .nii.gz

Returns return the numpy array

3.17.3 H5 Loader

Loads h5 files and some associated information

class deepreg.dataset.loader.h5_loader.**H5FileLoader**(*dir_paths: List[str]*, *name: str*, *grouped: bool*)

Generalized loader for h5 files

Parameters

- **dir_paths** – path to the directory of the data set
- **name** – name is used to identify the subdirectories or file names
- **grouped** – true if the data is grouped

close()

Close opened h5 file handles

get_data(*index: (<class 'int'>, <class 'tuple'>)*) → numpy.ndarray

Get one data array by specifying an index

Parameters **index** – the data index which is required

- for paired or unpaired, the index is one single int, data_index
- for grouped, the index is a tuple of two ints, (group_index, in_group_data_index)

Returns arr the data array at the specified index

get_data_ids()

Return the unique IDs of the data in this data set this function is used to verify the consistency between images and label, moving and fixed

Returns data_path_splits as the data can be identified using dir_path and data_key

get_num_images() → int

Returns int, number of images in this data set

set_data_structure()

Store the data structure in the memory so that we can retrieve data using data_index this function sets two attributes

- h5_files, a dict such that h5_files[dir_path] = opened h5 file handle
- data_path_splits, a list of string tuples to identify path of data
  - if grouped, a split is (dir_path, group_name, data_key) such that data = h5_files[dir_path]["group-{group_name}-{data_key}"]
  - if not grouped, a split is (dir_path, data_key) such that data = h5_files[dir_path][data_key]

set_group_structure()

Same code as NiftiLoader, as the first two tokens of a split forms a group_id

In addition to set_data_structure store the group structure in the group_struct so that group_struct[group_index] = list of data_index we can retrieve data using (group_index, in_group_data_index) data_index = group_struct[group_index][in_group_data_index]

3.18 Network

3.18.1 DDF / DVF Network

deepreg.model.network.ddf_dvf.**build_ddf_dvf_model**(*moving_image_size: tuple*, *fixed_image_size: tuple*, *index_size: int*, *labeled: bool*, *batch_size: int*, *model_config: dict*, *loss_config: dict*) → tensorflow.keras.Model

Build a model which outputs DDF/DVF.

Parameters

- **moving_image_size** – (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – (f_dim1, f_dim2, f_dim3)
- **index_size** – int, the number of indices for identifying a sample
- **labeled** – bool, indicating if the data is labeled
- **batch_size** – int, size of mini-batch
- **model_config** – config for the model
- **loss_config** – config for the loss

Returns the built tf.keras.Model

deepreg.model.network.ddf_dvf.**ddf_dvf_forward**(*backbone: tensorflow.keras.Model*, *moving_image: tensorflow.Tensor*, *fixed_image: tensorflow.Tensor*, *moving_label: tensorflow.Tensor, None*, *moving_image_size: tuple*, *fixed_image_size: tuple*, *output_dvf: bool*) → [tensorflow.Tensor, None, tensorflow.Tensor, tensorflow.Tensor, tensorflow.Tensor, None, tensorflow.Tensor]

Perform the network forward pass.

Parameters

- **backbone** – model architecture object, e.g. model.backbone.local_net
- **moving_image** – tensor of shape (batch, m_dim1, m_dim2, m_dim3)
- **fixed_image** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **moving_label** – tensor of shape (batch, m_dim1, m_dim2, m_dim3) or None
- **moving_image_size** – tuple like (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – tuple like (f_dim1, f_dim2, f_dim3)
- **output_dvf** – bool, if true, model outputs dvf, if false, model outputs ddf

Returns

(dvf, ddf, pred_fixed_image, pred_fixed_label, fixed_grid), where

- dvf is the dense velocity field of shape (batch, f_dim1, f_dim2, f_dim3, 3)
- ddf is the dense displacement field of shape (batch, f_dim1, f_dim2, f_dim3, 3)

- pred_fixed_image is the predicted (warped) moving image of shape (batch, f_dim1, f_dim2, f_dim3)
- pred_fixed_label is the predicted (warped) moving label of shape (batch, f_dim1, f_dim2, f_dim3)
- fixed_grid is the grid of shape(f_dim1, f_dim2, f_dim3, 3)

3.18.2 Conditional Network

deepreg.model.network.cond.**build_conditional_model**(*moving_image_size: tuple*, *fixed_image_size: tuple*, *index_size: int*, *labeled: bool*, *batch_size: int*, *model_config: dict*, *loss_config: dict*) → tensorflow.keras.Model

Build a model which outputs predicted fixed label.

Parameters

- **moving_image_size** – (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – (f_dim1, f_dim2, f_dim3)
- **index_size** – int, the number of indices for identifying a sample
- **labeled** – bool, indicating if the data is labeled
- **batch_size** – int, size of mini-batch
- **model_config** – config for the model
- **loss_config** – config for the loss

Returns the built tf.keras.Model

deepreg.model.network.cond.**conditional_forward**(*backbone: tensorflow.keras.Model*, *moving_image: tensorflow.Tensor*, *fixed_image: tensorflow.Tensor*, *moving_label: tensorflow.Tensor, None*, *moving_image_size: tuple*, *fixed_image_size: tuple*) → [tensorflow.Tensor, tensorflow.Tensor]

Perform the network forward pass.

Parameters

- **backbone** – model architecture object, e.g. model.backbone.local_net
- **moving_image** – tensor of shape (batch, m_dim1, m_dim2, m_dim3)
- **fixed_image** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **moving_label** – tensor of shape (batch, m_dim1, m_dim2, m_dim3) or None
- **moving_image_size** – tuple like (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – tuple like (f_dim1, f_dim2, f_dim3)

Returns

(pred_fixed_label, fixed_grid), where

- pred_fixed_label is the predicted (warped) moving label of shape (batch, f_dim1, f_dim2, f_dim3)

- fixed_grid is the grid of shape(f_dim1, f_dim2, f_dim3, 3)

3.18.3 Affine Network

deepreg.model.network.affine.**affine_forward**(*backbone: tensorflow.keras.Model*, *moving_image: tensorflow.Tensor*, *fixed_image: tensorflow.Tensor*, *moving_label: tensorflow.Tensor, None*, *moving_image_size: tuple*, *fixed_image_size: tuple*)

Perform the network forward pass.

Parameters

- **backbone** – model architecture object, e.g. model.backbone.local_net
- **moving_image** – tensor of shape (batch, m_dim1, m_dim2, m_dim3)
- **fixed_image** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **moving_label** – tensor of shape (batch, m_dim1, m_dim2, m_dim3) or None
- **moving_image_size** – tuple like (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – tuple like (f_dim1, f_dim2, f_dim3)

Returns

tuple(affine, ddf, pred_fixed_image, pred_fixed_label, fixed_grid), where

- affine is the affine transformation matrix predicted by the network (batch, 4, 3)
- ddf is the dense displacement field of shape (batch, f_dim1, f_dim2, f_dim3, 3)
- pred_fixed_image is the predicted (warped) moving image of shape (batch, f_dim1, f_dim2, f_dim3)
- pred_fixed_label is the predicted (warped) moving label of shape (batch, f_dim1, f_dim2, f_dim3)
- fixed_grid is the grid of shape(f_dim1, f_dim2, f_dim3, 3)

deepreg.model.network.affine.**build_affine_model**(*moving_image_size: tuple*, *fixed_image_size: tuple*, *index_size: int*, *labeled: bool*, *batch_size: int*, *model_config: dict*, *loss_config: dict*)

Build a model which outputs the parameters for affine transformation.

Parameters

- **moving_image_size** – (m_dim1, m_dim2, m_dim3)
- **fixed_image_size** – (f_dim1, f_dim2, f_dim3)
- **index_size** – int, the number of indices for identifying a sample
- **labeled** – bool, indicating if the data is labeled
- **batch_size** – int, size of mini-batch
- **model_config** – config for the model
- **loss_config** – config for the loss

Returns the built tf.keras.Model

3.18.4 Utils

deepreg.model.network.build.**build_model**(*moving_image_size: tuple*, *fixed_image_size: tuple*, *index_size: int*, *labeled: bool*, *batch_size: int*, *model_config: dict*, *loss_config: dict*)

Parsing algorithm types to model building functions.

Parameters

- **moving_image_size** – [m_dim1, m_dim2, m_dim3]
- **fixed_image_size** – [f_dim1, f_dim2, f_dim3]
- **index_size** – dataset size
- **labeled** – true if the label of moving/fixed images are provided
- **batch_size** – mini-batch size
- **model_config** – model configuration, e.g. dictionary return from parser.yaml.load
- **loss_config** – loss configuration, e.g. dictionary return from parser.yaml.load

Returns the built tf.keras.Model

Module to build backbone modules based on passed inputs.

deepreg.model.network.util.**add_ddf_loss**(*model: tensorflow.keras.Model*, *ddf: tensorflow.Tensor*, *loss_config: dict*) → tensorflow.keras.Model

Add regularization loss of ddf into model.

Parameters

- **model** – tf.keras.Model
- **ddf** – tensor of shape (batch, m_dim1, m_dim2, m_dim3, 3)
- **loss_config** – config for loss

deepreg.model.network.util.**add_image_loss**(*model: tensorflow.keras.Model*, *fixed_image: tensorflow.Tensor*, *pred_fixed_image: tensorflow.Tensor*, *loss_config: dict*) → tensorflow.keras.Model

Add image dissimilarity loss of ddf into model.

Parameters

- **model** – tf.keras.Model
- **fixed_image** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **pred_fixed_image** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **loss_config** – config for loss

deepreg.model.network.util.**add_label_loss**(*model: tensorflow.keras.Model*, *grid_fixed: tensorflow.Tensor*, *fixed_label: tensorflow.Tensor, None*, *pred_fixed_label: tensorflow.Tensor, None*, *loss_config: dict*) → tensorflow.keras.Model

Add label dissimilarity loss of ddf into model.

Parameters

- **model** – tf.keras.Model
- **grid_fixed** – tensor of shape (f_dim1, f_dim2, f_dim3, 3)

- **fixed_label** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **pred_fixed_label** – tensor of shape (batch, f_dim1, f_dim2, f_dim3)
- **loss_config** – config for loss

deepreg.model.network.util.**build_backbone**(*image_size: tuple*, *out_channels: int*, *model_config: dict*, *method_name: str*) → tensorflow.keras.Model

Backbone model accepts a single input of shape (batch, dim1, dim2, dim3, ch_in) and returns a single output of shape (batch, dim1, dim2, dim3, ch_out).

Parameters

- **image_size** – tuple, dims of image, (dim1, dim2, dim3)
- **out_channels** – int, number of out channels, ch_out
- **method_name** – str, one of ddf, dvf and conditional
- **model_config** – dict, model configuration, returned from parser.yaml.load

Returns tf.keras.Model

deepreg.model.network.util.**build_inputs**(*moving_image_size: tuple*, *fixed_image_size: tuple*, *index_size: int*, *batch_size: int*, *labeled: bool*) → [tensorflow.keras.Input, tensorflow.keras.Input, tensorflow.keras.Input, tensorflow.keras.Input, tensorflow.keras.Input]

Configure a pair of moving and fixed images and a pair of moving and fixed labels as model input and returns model input tf.keras.Input.

Parameters

- **moving_image_size** – tuple, dims of moving images, [m_dim1, m_dim2, m_dim3]
- **fixed_image_size** – tuple, dims of fixed images, [f_dim1, f_dim2, f_dim3]
- **index_size** – int, dataset size (number of images)
- **batch_size** – int, mini-batch size
- **labeled** – Boolean, true if we have label data

Returns 5 (if labeled=True) or 3 (if labeled=False) tf.keras.Input objects

3.19 Network Backbone

3.19.1 Local Net

class deepreg.model.backbone.local_net.**LocalNet**(**args: Any*, ***kwargs: Any*)

Builds LocalNet for image registration.

Reference:

- Hu, Yipeng, et al. "Weakly-supervised convolutional neural networks for multimodal image registration." Medical image analysis 49 (2018): 1-13. https://doi.org/10.1016/j.media.2018.07.002
- Hu, Yipeng, et al. "Label-driven weakly-supervised learning for multimodal deformable image registration," https://arxiv.org/abs/1711.01666

Image is encoded gradually, i from level 0 to E, then it is decoded gradually, j from level E to D. Some of the decoded levels are used for generating extractions

So, extract_levels are between [0, E] with E = max(extract_levels), and D = min(extract_levels).

Parameters

- **image_size** – tuple, such as (dim1, dim2, dim3)
- **out_channels** – int, number of channels for the extractions
- **num_channel_initial** – int, number of initial channels.
- **extract_levels** – list of int, number of extraction levels.
- **out_kernel_initializer** – str, initializer to use for kernels.
- **out_activation** – str, activation to use at end layer.
- **kwargs** –

call(*inputs*, *training=None*, *mask=None*)
Build LocalNet graph based on built layers.

Parameters

- **inputs** – image batch, shape = (batch, f_dim1, f_dim2, f_dim3, ch)
- **training** – None or bool.
- **mask** – None or tf.Tensor.

Returns tf.Tensor, shape = (batch, f_dim1, f_dim2, f_dim3, out_channels)

3.19.2 Global Net

class deepreg.model.backbone.global_net.**GlobalNet**(**args: Any*, ***kwargs: Any*)
Builds GlobalNet for image registration.

Reference:

- Hu, Yipeng, et al. "Label-driven weakly-supervised learning for multimodal deformable image registration," https://arxiv.org/abs/1711.01666

Image is encoded gradually, i from level 0 to E. Then, a densely-connected layer outputs an affine transformation.

Parameters

- **image_size** – tuple, such as (dim1, dim2, dim3)
- **out_channels** – int, number of channels for the output
- **num_channel_initial** – int, number of initial channels
- **extract_levels** – list, which levels from net to extract
- **out_kernel_initializer** – str, which kernel to use as initializer
- **out_activation** – str, activation at last layer
- **kwargs** –

call(*inputs*, *training=None*, *mask=None*)
Build GlobalNet graph based on built layers.

Parameters

- **inputs** – image batch, shape = (batch, f_dim1, f_dim2, f_dim3, ch)

- **training** – None or bool.
- **mask** – None or tf.Tensor.

Returns tf.Tensor, shape = (batch, dim1, dim2, dim3, 3)

3.19.3 U-Net

class deepreg.model.backbone.u_net.**UNet**(**args: Any*, ***kwargs: Any*)

Class that implements an adapted 3D UNet.

Reference:

- O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation,", Lecture Notes in Computer Science, 2015, vol. 9351, pp. 234–241. https://arxiv.org/abs/1505.04597

Initialise UNet.

Parameters

- **image_size** – tuple, (dim1, dim2, dim3), dims of input image.
- **out_channels** – int, number of channels for the output
- **num_channel_initial** – int, number of initial channels
- **depth** – int, input is at level 0, bottom is at level depth
- **out_kernel_initializer** – str, which kernel to use as initializer
- **out_activation** – str, activation at last layer
- **pooling** – Boolean, for downsampling, use non-parameterized pooling if true, otherwise use conv3d
- **concat_skip** – Boolean, when upsampling, concatenate skipped tensor if true, otherwise use addition
- **kwargs** –

call(*inputs*, *training=None*, *mask=None*)

Builds graph based on built layers.

Parameters

- **inputs** – shape = [batch, f_dim1, f_dim2, f_dim3, in_channels]
- **training** –
- **mask** –

Returns shape = [batch, f_dim1, f_dim2, f_dim3, out_channels]

3.20 Layer

3.20.1 Layer

Activation

class deepreg.model.layer.**Activation**(**args: Any*, ***kwargs: Any*)
Layer wraps tf.keras.activations.get().

Parameters

- **identifier** – e.g. "relu"
- **kwargs** –

AdditiveUpSampling

class deepreg.model.layer.**AdditiveUpSampling**(**args: Any*, ***kwargs: Any*)
Layer up-samples 3d tensor and reduce channels using split and sum.

Parameters

- **output_shape** – (out_dim1, out_dim2, out_dim3)
- **strides** – int, 1-D Tensor or list
- **kwargs** –

call(*inputs*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, dim1, dim2, dim3, channels)
- **kwargs** –

Returns shape = (batch, out_dim1, out_dim2, out_dim3, channels//stride]

Conv3d

class deepreg.model.layer.**Conv3d**(**args: Any*, ***kwargs: Any*)
Layer wraps tf.keras.layers.Conv3D.

Parameters

- **filters** – number of channels of the output
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **padding** – str, same or valid
- **activation** – str, defines the activation function
- **use_bias** – bool, whether add bias to output
- **kernel_initializer** – str, defines the initialization method, defines the initialization method

Conv3dBlock

class deepreg.model.layer.**Conv3dBlock**(**args: Any*, ***kwargs: Any*)
A conv3d block having conv3d - norm - activation.

Parameters

- **filters** – number of channels of the output
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **padding** – str, same or valid

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, in_dim1, in_dim2, in_dim3, channels)
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Returns shape = (batch, in_dim1, in_dim2, in_dim3, channels)

Conv3dWithResize

class deepreg.model.layer.**Conv3dWithResize**(**args: Any*, ***kwargs: Any*)
A layer contains conv3d - resize3d.

Parameters

- **output_shape** – tuple, (out_dim1, out_dim2, out_dim3)
- **filters** – int, number of channels of the output
- **kernel_initializer** – str, defines the initialization method
- **activation** – str, defines the activation function
- **kwargs** –

call(*inputs*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, dim1, dim2, dim3, channels)
- **kwargs** –

Returns shape = (batch, out_dim1, out_dim2, out_dim3, channels)

Deconv3d

class deepreg.model.layer.**Deconv3d**(**args: Any*, ***kwargs: Any*)
Layer wraps tf.keras.layers.Conv3DTranspose and does not requires input shape when initializing.

Parameters

- **filters** – number of channels of the output
- **output_shape** – (out_dim1, out_dim2, out_dim3)
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **padding** – str, same or valid
- **kwargs** –

Deconv3dBlock

class deepreg.model.layer.**Deconv3dBlock**(**args: Any*, ***kwargs: Any*)
A deconv3d block having deconv3d - norm - activation.

Parameters

- **filters** – number of channels of the output
- **output_shape** – (out_dim1, out_dim2, out_dim3)
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **padding** – str, same or valid
- **kwargs** –

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, in_dim1, in_dim2, in_dim3, channels)
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Return output shape = (batch, in_dim1, in_dim2, in_dim3, channels)

Dense

class deepreg.model.layer.**Dense**(**args: Any*, ***kwargs: Any*)
Layer wraps tf.keras.layers.Dense and flattens input if necessary.

Parameters

- **units** – number of hidden units
- **bias_initializer** – str, default "zeros"
- **kwargs** –

call(*inputs*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, *vol_dim, channels)
- **kwargs** – (not used)

Returns shape = (batch, units)

DownSampleResnetBlock

class deepreg.model.layer.**DownSampleResnetBlock**(**args: Any*, ***kwargs: Any*)

A down-sampling resnet conv3d block, with max-pooling or conv3d.

1. conved = conv3d_block(inputs) # adjust channel
2. skip = residual_block(conved) # develop feature
3. pooled = pool(skip) # down-sample

Parameters

- **filters** – number of channels of the output
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **padding** – str, same or valid

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, in_dim1, in_dim2, in_dim3, channels)
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Returns

(pooled, skip)

- downsampled, shape = (batch, in_dim1//2, in_dim2//2, in_dim3//2, channels)
- skipped, shape = (batch, in_dim1, in_dim2, in_dim3, channels)

IntDVF

class deepreg.model.layer.**IntDVF**(**args: Any*, ***kwargs: Any*)

Layer calculates DVF from DDF.

Reference:

- integrate_vec of neuron https://github.com/adalca/neurite/blob/legacy/neuron/utils.py

Parameters

- **fixed_image_size** – tuple, (f_dim1, f_dim2, f_dim3)
- **num_steps** – int, number of steps for integration
- **kwargs** –

call(*inputs*, ***kwargs*)

Parameters

- **inputs** – dvf, shape = (batch, f_dim1, f_dim2, f_dim3, 3), type = float32
- **kwargs** –

Returns ddf, shape = (batch, f_dim1, f_dim2, f_dim3, 3)

LocalNetResidual3dBlock

class deepreg.model.layer.**LocalNetResidual3dBlock**(**args: Any*, ***kwargs: Any*)

A resnet conv3d block, simpler than Residual3dBlock.

1. conved = conv3d(inputs)
2. out = act(norm(conved) + inputs)

Parameters

- **filters** – number of channels of the output
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **kwargs** –

LocalNetUpSampleResnetBlock

class deepreg.model.layer.**LocalNetUpSampleResnetBlock**(**args: Any*, ***kwargs: Any*)

Layer up-samples tensor with two inputs (skipped and down-sampled).

Parameters

- **filters** – int, number of output channels
- **use_additive_upsampling** – bool to used additive upsampling (default is True)
- **kwargs** –

build(*input_shape*)

Parameters **input_shape** – tuple (nonskip_tensor_shape, skip_tensor_shape)

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – list = [inputs_nonskip, inputs_skip]
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Returns

MaxPool3d

class deepreg.model.layer.**MaxPool3d**(**args: Any*, ***kwargs: Any*)
Layer wraps tf.keras.layers.MaxPool3D

Parameters

- **pool_size** – int or tuple of 3 ints
- **strides** – int or tuple of 3 ints or None, if None default will be pool_size
- **padding** – str, same or valid
- **kwargs** –

Norm

class deepreg.model.layer.**Norm**(**args: Any*, ***kwargs: Any*)
Class merges batch norm and layer norm.

Parameters

- **name** – str, batch_norm or layer_norm
- **axis** – int
- **kwargs** –

Residual3dBlock

class deepreg.model.layer.**Residual3dBlock**(**args: Any*, ***kwargs: Any*)
A resnet conv3d block.

1. conved = conv3d(conv3d_block(inputs))
2. out = act(norm(conved) + inputs)

Parameters

- **filters** – int, number of filters in the convolutional layers
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **strides** – int or tuple of 3 ints, e.g. (1,1,1) or 1
- **kwargs** –

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – shape = (batch, in_dim1, in_dim2, in_dim3, channels)
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Return output shape = (batch, in_dim1, in_dim2, in_dim3, channels)

UpSampleResnetBlock

class deepreg.model.layer.**UpSampleResnetBlock**(**args: Any*, ***kwargs: Any*)
An up-sampling resnet conv3d block, with deconv3d.

Parameters

- **filters** – number of channels of the output
- **kernel_size** – int or tuple of 3 ints, e.g. (3,3,3) or 3
- **concat** – bool,specify how to combine input and skip connection images. If True, use concatenation if false use sum (default=False).
- **kwargs** –

build(*input_shape*)

Parameters **input_shape** – tuple, (downsampled_image_shape, skip_connection image_shape)

call(*inputs*, *training=None*, ***kwargs*)

Parameters

- **inputs** – tuple
  - down-sampled
  - skipped
- **training** – training flag for normalization layers (default: None)
- **kwargs** –

Returns shape = (batch, *skip_connection_image_shape, filters]

Warping

class deepreg.model.layer.**Warping**(**args: Any*, ***kwargs: Any*)
A layer warps an image using DDF.

Reference:

- transform of neuron https://github.com/adalca/neurite/blob/legacy/neuron/utils.py

  where vol = image, loc_shift = ddf

Parameters

- **fixed_image_size** – shape = (f_dim1, f_dim2, f_dim3) or (f_dim1, f_dim2, f_dim3, ch) with the last channel for features
- **kwargs** –

call(*inputs*, ***kwargs*)

Parameters

- **inputs** – (ddf, image)
  - ddf, shape = (batch, f_dim1, f_dim2, f_dim3, 3), dtype = float32
  - image, shape = (batch, m_dim1, m_dim2, m_dim3), dtype = float32
- **kwargs** –

Returns shape = (batch, f_dim1, f_dim2, f_dim3)

3.20.2 Util

Module containing utilities for layer inputs

deepreg.model.layer_util.**get_n_bits_combinations**(*num_bits: int*) → list

Function returning list containing all combinations of n bits. Given num_bits binary bits, each bit has value 0 or 1, there are in total 2**n_bits combinations.

Parameters **num_bits** – int, number of combinations to evaluate

Returns a list of length 2**n_bits, return[i] is the binary representation of the decimal integer.

Example

```
>>> from deepreg.model.layer_util import get_n_bits_combinations
>>> get_n_bits_combinations(3)
[[0, 0, 0], # 0
 [0, 0, 1], # 1
 [0, 1, 0], # 2
 [0, 1, 1], # 3
 [1, 0, 0], # 4
 [1, 0, 1], # 5
 [1, 1, 0], # 6
 [1, 1, 1]] # 7
```

deepreg.model.layer_util.**get_reference_grid**(*grid_size: (<class 'tuple'>, <class 'list'>)*) → tensorflow.Tensor

Generate a 3D grid with given size.

Reference:

- volshape_to_meshgrid of neuron https://github.com/adalca/neurite/blob/legacy/neuron/utils.py

  neuron modifies meshgrid to make it faster, however local benchmark suggests tf.meshgrid is better

Note:

for tf.meshgrid, in the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing.

Parameters **grid_size** – list or tuple of size 3, [dim1, dim2, dim3]

Returns shape = [dim1, dim2, dim3, 3], grid[i, j, k, :] = [i j k]

deepreg.model.layer_util.**pyramid_combination**(*values: list*, *weights: list*) → tensorflow.Tensor

Calculates linear interpolation (a weighted sum) using values of hypercube corners in dimension n.

For example, when num_dimension = len(loc_shape) = num_bits = 3 values correspond to values at corners of following coordinates

```
[[0, 0, 0], # even
 [0, 0, 1], # odd
 [0, 1, 0], # even
 [0, 1, 1], # odd
 [1, 0, 0], # even
 [1, 0, 1], # odd
 [1, 1, 0], # even
 [1, 1, 1]] # odd
```

values[::2] correspond to the corners with last coordinate == 0

```
[[0, 0, 0],
 [0, 1, 0],
 [1, 0, 0],
 [1, 1, 0]]
```

values[1::2] correspond to the corners with last coordinate == 1

```
[[0, 0, 1],
 [0, 1, 1],
 [1, 0, 1],
 [1, 1, 1]]
```

The weights correspond to the floor corners. For example, when num_dimension = len(loc_shape) = num_bits = 3, weights = [w1, w2, w3] (ignoring the batch dimension).

So for corner with coords (x, y, z), x, y, z's values are 0 or 1

- weight for x = w1 if x = 0 else 1-w1
- weight for y = w2 if y = 0 else 1-w2
- weight for z = w3 if z = 0 else 1-w3

so the weight for (x, y, z) is

W_xyz = ((1-x) * w1 + x * (1-w1)) * ((1-y) * w2 + y * (1-w2)) * ((1-z) * w3 + z * (1-w3))

= (W_xy * (1-z)) * w3 + (W_xy * z) * (1-w3)

where W_xy is the weight for (x, y), let

- W_xy0 = W_xy * w3
- W_xy1 = W_xy * (1-w3)

So, the final sum V equals

sum over x,y,z (V_xyz * W_xyz)

= sum over x,y (V_xy0 * W_xy0 + V_xy1 * W_xy1)

= sum over x,y (V_xy0 * W_xy * w3 + V_xy1 * W_xy * (1-w3))

= sum over x,y (W_xy * (V_xy0 * w3 + V_xy1 * W_xy * (1-w3)))

That's why we call this pyramid combination. It calculates the linear interpolation gradually, starting from the last dimension. The key is that the weight of each corner is the product of the weights along each dimension.

Parameters

- **values** – a list having values on the corner, it has 2**n tensors of shape (*loc_shape) or (batch, *loc_shape) or (batch, *loc_shape, ch) the order is consistent with get_n_bits_combinations loc_shape is independent from n, aka num_dim
- **weights** – a list having weights of floor points, it has n tensors of shape (*loc_shape) or (batch, *loc_shape) or (batch, *loc_shape, 1)

Returns one tensor of the same shape as an element in values (*loc_shape) or (batch, *loc_shape) or (batch, *loc_shape, 1)

deepreg.model.layer_util.**random_transform_generator**(*batch_size: int*, *scale: float*, *seed: (<class 'int'>, None) = None*) → tensorflow.Tensor

Function that generates a random 3D transformation parameters for a batch of data.

for 3D coordinates, affine transformation is

```
[[x' y' z' 1]] = [[x y z 1]] * [[* * * 0]
                                [* * * 0]
                                [* * * 0]
                                [* * * 1]]
```

where each * represents a degree of freedom, so there are in total 12 degrees of freedom the equation can be denoted as

new = old * T

where

- new is the transformed coordinates, of shape (1, 4)
- old is the original coordinates, of shape (1, 4)
- T is the transformation matrix, of shape (4, 4)

the equation can be simplified to

```
[[x' y' z']] = [[x y z 1]] * [[* * *]
                              [* * *]
                              [* * *]
                              [* * *]]
```

so that

new = old * T

where

- new is the transformed coordinates, of shape (1, 3)
- old is the original coordinates, of shape (1, 4)
- T is the transformation matrix, of shape (4, 3)

Given original and transformed coordinates, we can calculate the transformation matrix using

x = np.linalg.lstsq(a, b)

such that

a x = b

In our case,

- a = old
- b = new
- x = T

To generate random transformation, we choose to add random perturbation to corner coordinates as follows: for corner of coordinates (x, y, z), the noise is

-(x, y, z) .* (r1, r2, r3)

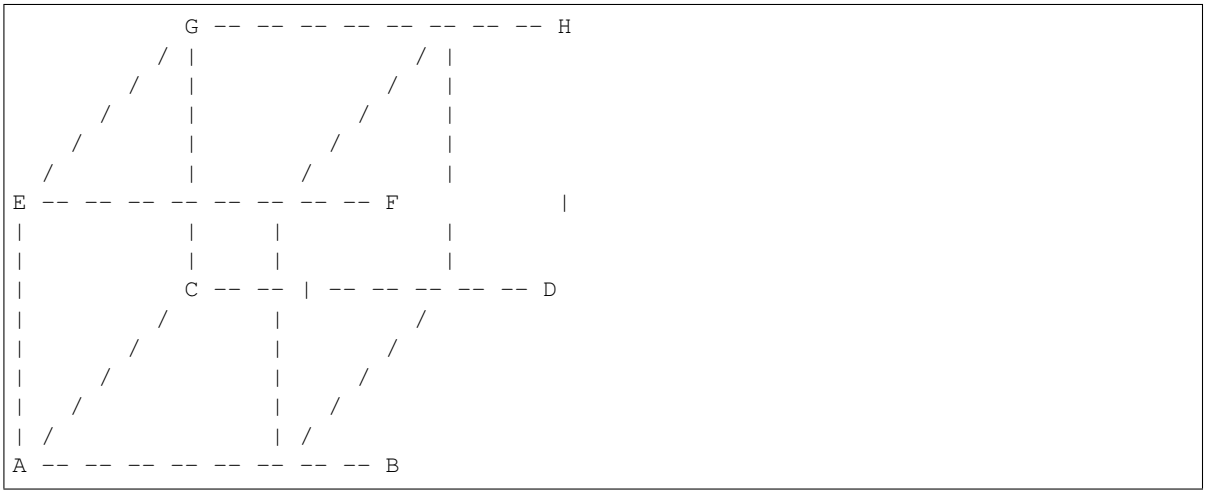
where ri is a random number between (0, scale). So

(x', y', z') = (x, y, z) .* (1-r1, 1-r2, 1-r3)

Thus, we can directly sample between 1-scale and 1 instead

We choose to calculate the transformation based on four corners in a cube centered at (0, 0, 0). A cube is shown as below, where

- C = (-1, -1, -1)
- G = (-1, -1, 1)
- D = (-1, 1, -1)
- A = (1, -1, -1)



Parameters

- **batch_size** – int
- **scale** – a float number between 0 and 1
- **seed** – control the randomness

Returns shape = (batch, 4, 3)

deepreg.model.layer_util.**resample**(*vol*, *loc*, *interpolation='linear'*)

Sample the volume at given locations.

Input has

- volume, vol, of shape = (batch, v_dim 1, ..., v_dim n), or (batch, v_dim 1, ..., v_dim n, ch), where n is the dimension of volume, ch is the extra dimension as features.

  Denote vol_shape = (v_dim 1, ..., v_dim n)

- location, loc, of shape = (batch, l_dim 1, ..., l_dim m, n), where m is the dimension of output.

  Denote loc_shape = (l_dim 1, ..., l_dim m)

Reference:

- neuron's interpn https://github.com/adalca/neurite/blob/legacy/neuron/utils.py

  Difference

  1. they dont have batch size
  2. they support more dimensions in vol

  TODO try not using stack as neuron claims it's slower

Parameters

- **vol** – shape = (batch, *vol_shape) or (batch, *vol_shape, ch) with the last channel for features
- **loc** – shape = (batch, *loc_shape, n) such that loc[b, l1, ..., ln, :] = [v1, ..., vn] is of shape (n,), which represents a point in vol, with coordinates (v1, ..., vn)
- **interpolation** – linear only, TODO support nearest

Returns shape = (batch, l_dim 1, ..., l_dim n)

deepreg.model.layer_util.**resize3d**(*image: tensorflow.Tensor*, *size: (<class 'tuple'>, <class 'list'>)*, *method: str = tensorflow.image.ResizeMethod.BILINEAR*) → tensorflow.Tensor

Tensorflow does not have resize 3d, therefore the resize is performed two folds.

- resize dim2 and dim3
- resize dim1 and dim2

Parameters

- **image** – tensor of shape = (batch, dim1, dim2, dim3, channels) or (batch, dim1, dim2, dim3) or (dim1, dim2, dim3)
- **size** – tuple, (out_dim1, out_dim2, out_dim3)
- **method** – str, one of tf.image.ResizeMethod

Returns tensor of shape = (batch, out_dim1, out_dim2, out_dim3, channels) or (batch, dim1, dim2, dim3) or (dim1, dim2, dim3)

deepreg.model.layer_util.**warp_grid**(*grid: tensorflow.Tensor*, *theta: tensorflow.Tensor*) → tensorflow.Tensor

Perform transformation on the grid.

- grid_padded[i,j,k,:] = [i j k 1]
- grid_warped[b,i,j,k,p] = sum_over_q (grid_padded[i,j,k,q] * theta[b,q,p])

Parameters

- **grid** – shape = (dim1, dim2, dim3, 3), grid[i,j,k,:] = [i j k]
- **theta** – parameters of transformation, shape = (batch, 4, 3)

Returns shape = (batch, dim1, dim2, dim3, 3)

deepreg.model.layer_util.**warp_image_ddf**(*image: tensorflow.Tensor*, *ddf: tensorflow.Tensor*, *grid_ref: tensorflow.Tensor, None*) → tensorflow.Tensor

Warp an image with given DDF.

Parameters

- **image** – an image to be warped, shape = (batch, m_dim1, m_dim2, m_dim3) or (batch, m_dim1, m_dim2, m_dim3, ch)
- **ddf** – shape = (batch, f_dim1, f_dim2, f_dim3, 3)
- **grid_ref** – shape = (1, f_dim1, f_dim2, f_dim3, 3) or None, if None grid_reg will be calculated based on ddf

Returns shape = (batch, f_dim1, f_dim2, f_dim3) or (batch, f_dim1, f_dim2, f_dim3, ch)

3.21 Loss

3.21.1 Image Loss

Module provides different loss functions for calculating the dissimilarities between images.

deepreg.model.loss.image.**dissimilarity_fn**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *name: str*, ***kwargs*) → tensorflow.Tensor

Returns the calculated dissimilarity for each batch.

Parameters

- **y_true** – fixed_image, shape = (batch, f_dim1, f_dim2, f_dim3)
- **y_pred** – warped_moving_image, shape = (batch, f_dim1, f_dim2, f_dim3)
- **name** – name of the dissimilarity function
- **kwargs** – absorb additional parameters

Returns shape = (batch,)

deepreg.model.loss.image.**global_mutual_information**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *num_bins: int = 23*, *sigma_ratio: float = 0.5*) → tensorflow.Tensor

Differentiable global mutual information loss via Parzen windowing method.

Reference: https://dspace.mit.edu/handle/1721.1/123142, Section 3.1, equation 3.1-3.5, Algorithm 1

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3, ch)
- **y_pred** – shape = (batch, dim1, dim2, dim3, ch)
- **num_bins** – int, number of bins for intensity
- **sigma_ratio** – float, a hyper param for gaussian function

Returns shape = (batch,)

deepreg.model.loss.image.**local_normalized_cross_correlation**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *kernel_size: int = 9*, *kernel_type: str = 'rectangular'*, ***kwargs*) → tensorflow.Tensor

Local squared zero-normalized cross-correlation. The loss is based on a moving kernel/window over the y_true/y_pred, within the window the square of zncc is calculated. The kernel can be a rectangular / triangular / gaussian window. The final loss is the averaged loss over all windows.

Reference:

- Zero-normalized cross-correlation (ZNCC): https://en.wikipedia.org/wiki/Cross-correlation
- Code: https://github.com/voxelmorph/voxelmorph/blob/legacy/src/losses.py

TODO: is it possible to not using tf.nn.conv3d?

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3, ch)
- **y_pred** – shape = (batch, dim1, dim2, dim3, ch)
- **kernel_size** – int. Kernel size or kernel sigma for kernel_type='gauss'.
- **kernel_type** – str ('triangular', 'gaussian' default: 'rectangular')
- **kwargs** – absorb additional parameters

Returns shape = (batch,)

deepreg.model.loss.image.**ssd**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*) → tensorflow.Tensor

Sum of squared distance between y_true and y_pred.

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3, ch)
- **y_pred** – shape = (batch, dim1, dim2, dim3, ch)

Returns shape = (batch,)

3.21.2 Label Loss

Module provides different loss functions for calculating the dissimilarities between labels.

deepreg.model.loss.label.**cauchy_kernel1d**(*sigma: int*) → tensorflow.Tensor

Approximating cauchy kernel in 1d.

Parameters **sigma** – int, defining standard deviation of kernel.

Returns shape = (dim,) or ()

deepreg.model.loss.label.**compute_centroid**(*mask: tensorflow.Tensor*, *grid: tensorflow.Tensor*) → tensorflow.Tensor

Calculate the centroid of the mask.

Parameters

- **mask** – shape = (batch, dim1, dim2, dim3)
- **grid** – shape = (dim1, dim2, dim3, 3)

Returns shape = (batch, 3), batch of vectors denoting location of centroids.

deepreg.model.loss.label.**compute_centroid_distance**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *grid: tensorflow.Tensor*) → tensorflow.Tensor

Calculate the L2-distance between two tensors' centroids.

Parameters

- **y_true** – tensor, shape = (batch, dim1, dim2, dim3)
- **y_pred** – tensor, shape = (batch, dim1, dim2, dim3)
- **grid** – tensor, shape = (dim1, dim2, dim3, 3)

Returns shape = (batch,)

deepreg.model.loss.label.**dice_score**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *binary: bool = False*) → tensorflow.Tensor

Calculates dice score:

1. num = 2 * y_true * y_pred

2. denom = y_true + y_pred
3. dice score = num / denom

where num and denom are summed over the entire image first.

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3)
- **y_pred** – shape = (batch, dim1, dim2, dim3)
- **binary** – True if the y should be projected to 0 or 1

Returns shape = (batch,)

deepreg.model.loss.label.**dice_score_generalized**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *pos_weight: float = 1*, *neg_weight: float = 0*) → tensorflow.Tensor

Calculates weighted dice score:

1. let y_prod = y_true * y_pred and y_sum = y_true + y_pred
2. num = 2 * (pos_w * y_true * y_pred + neg_w * (1y_true) * (1y_pred))

   = 2 * ((pos_w+neg_w) * y_prod - neg_w * y_sum + neg_w)
3. denom = (pos_w * (y_true + y_pred) + neg_w * (1y_true + 1y_pred))

   = (pos_w-neg_w) * y_sum + 2 * neg_w
4. dice score = num / denom

where num and denom are summed over the entire image first.

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3)
- **y_pred** – shape = (batch, dim1, dim2, dim3)
- **pos_weight** – weight of positive class, default = 1
- **neg_weight** – weight of negative class, default = 0

Returns shape = (batch,)

deepreg.model.loss.label.**foreground_proportion**(*y: tensorflow.Tensor*) → tensorflow.Tensor

Calculating the percentage of foreground vs background per 3d volume.

Parameters **y** – shape = (batch, dim1, dim2, dim3), a 3D label tensor

Returns shape = (batch,)

deepreg.model.loss.label.**gauss_kernel1d**(*sigma: int*) → tensorflow.Tensor

Calculates a gaussian kernel.

Parameters **sigma** – number defining standard deviation for gaussian kernel.

Returns shape = (dim,) or ()

deepreg.model.loss.label.**get_dissimilarity_fn**(*config: dict*) → Callable

Parse arguments from a configuration dictionary and return the loss by averaging batch loss returned by multi- or single-scale loss functions.

Parameters **config** – dict, containing configuration for training.

Returns loss function, which returns a tensor of shape (batch,)

deepreg.model.loss.label.**jaccard_index**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*) → tensorflow.Tensor

Calculates jaccard index:

1. num = y_true * y_pred
2. denom = y_true + y_pred - y_true * y_pred
3. jaccard index = num / denom

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3)
- **y_pred** – shape = (batch, dim1, dim2, dim3)

Returns shape = (batch,)

deepreg.model.loss.label.**multi_scale_loss**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *loss_type: str*, *loss_scales: list*) → tensorflow.Tensor

Apply the loss at different scales (gaussian smoothing). It is assumed that loss values are between 0 and 1.

Parameters

- **y_true** – tensor, shape = (batch, dim1, dim2, dim3)
- **y_pred** – tensor, shape = (batch, dim1, dim2, dim3)
- **loss_type** – string, indicating which loss to pass to function single_scale_loss.

  Supported:

  – cross-entropy

  – mean-squared

  – dice

  – dice_generalized

  – jaccard
- **loss_scales** – list, values of sigma to pass to func gauss_kernel_1d.

Returns (batch,)

deepreg.model.loss.label.**separable_filter3d**(*tensor: tensorflow.Tensor*, *kernel: tensorflow.Tensor*) → tensorflow.Tensor

Creates a 3d separable filter.

Here *tf.nn.conv3d* accepts the *filters* argument of shape (filter_depth, filter_height, filter_width, in_channels, out_channels), where the first axis of *filters* is the depth not batch, and the input to *tf.nn.conv3d* is of shape (batch, in_depth, in_height, in_width, in_channels).

Parameters

- **tensor** – shape = (batch, dim1, dim2, dim3)
- **kernel** – shape = (dim4,)

Returns shape = (batch, dim1, dim2, dim3)

deepreg.model.loss.label.**single_scale_loss**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *loss_type: str*) → tensorflow.Tensor

Calculate the loss on two tensors based on defined loss.

Parameters

- **y_true** – tensor, shape = (batch, dim1, dim2, dim3)
- **y_pred** – tensor, shape = (batch, dim1, dim2, dim3)
- **loss_type** – string, indicating which loss to pass to function single_scale_loss.

  Supported:

  – cross-entropy

  – mean-squared

  – dice

  – dice_generalized

  – jaccard

Returns shape = (batch,)

deepreg.model.loss.label.**squared_error**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*) → tensorflow.Tensor

Calculates the mean squared difference between y_true, y_pred.

mean((y_true - y_pred)(y_true - y_pred))

Parameters

- **y_true** – tensor, shape = (batch, dim1, dim2, dim3)
- **y_pred** – shape = (batch, dim1, dim2, dim3)

Returns shape = (batch,)

deepreg.model.loss.label.**weighted_binary_cross_entropy**(*y_true: tensorflow.Tensor*, *y_pred: tensorflow.Tensor*, *pos_weight: float = 1*) → tensorflow.Tensor

Calculates weighted binary cross- entropy:

-loss = pos_w * y_true log(y_pred) - (1y_true) log(1y_pred)

Parameters

- **y_true** – shape = (batch, dim1, dim2, dim3)
- **y_pred** – shape = (batch, dim1, dim2, dim3)
- **pos_weight** – weight of positive class, scalar. Default value is 1

Returns shape = (batch,)

3.21.3 Deformation Loss

Module provides regularization energy functions for ddf.

deepreg.model.loss.deform.**compute_bending_energy**(*ddf: tensorflow.Tensor*) → tensorflow.Tensor

Calculate the bending energy based on second-order differentiation of ddf using central finite difference.

Parameters **ddf** – shape = (batch, m_dim1, m_dim2, m_dim3, 3)

Returns shape = (batch,)

deepreg.model.loss.deform.**compute_gradient_norm**(*ddf: tensorflow.Tensor*, *l1: bool = False*) → tensorflow.Tensor

Calculate the L1/L2 norm of the first-order differentiation of ddf using central finite difference.

Parameters

- **ddf** – shape = (batch, m_dim1, m_dim2, m_dim3, 3)
- **l1** – bool true if calculate L1 norm, otherwise L2 norm

Returns shape = (batch,)

deepreg.model.loss.deform.**gradient_dx**(*fx: tensorflow.Tensor*) → tensorflow.Tensor

Calculate gradients on x-axis of a 3D tensor using central finite difference. It moves the tensor along axis 1 to calculate the approximate gradient, the x axis, dx[i] = (x[i+1] - x[i-1]) / 2.

Parameters **fx** – shape = (batch, m_dim1, m_dim2, m_dim3)

Returns shape = (batch, m_dim1-2, m_dim2-2, m_dim3-2)

deepreg.model.loss.deform.**gradient_dxyz**(*fxyz: tensorflow.Tensor*, *fn: Callable*) → tensorflow.Tensor

Calculate gradients on x,y,z-axis of a tensor using central finite difference. The gradients are calculated along x, y, z separately then stacked together.

Parameters

- **fxyz** – shape = (..., 3)
- **fn** – function to call

Returns shape = (..., 3)

deepreg.model.loss.deform.**gradient_dy**(*fy: tensorflow.Tensor*) → tensorflow.Tensor

Calculate gradients on y-axis of a 3D tensor using central finite difference. It moves the tensor along axis 2 to calculate the approximate gradient, the y axis, dy[i] = (y[i+1] - y[i-1]) / 2.

Parameters **fy** – shape = (batch, m_dim1, m_dim2, m_dim3)

Returns shape = (batch, m_dim1-2, m_dim2-2, m_dim3-2)

deepreg.model.loss.deform.**gradient_dz**(*fz: tensorflow.Tensor*) → tensorflow.Tensor

Calculate gradients on z-axis of a 3D tensor using central finite difference. It moves the tensor along axis 3 to calculate the approximate gradient, the z axis, dz[i] = (z[i+1] - z[i-1]) / 2.

Parameters **fz** – shape = (batch, m_dim1, m_dim2, m_dim3)

Returns shape = (batch, m_dim1-2, m_dim2-2, m_dim3-2)

deepreg.model.loss.deform.**local_displacement_energy**(*ddf: tensorflow.Tensor*, *energy_type: str*, ***kwargs*) → tensorflow.Tensor

Calculate the displacement energy of the ddf based on finite difference.

Parameters

- **ddf** – shape = (batch, m_dim1, m_dim2, m_dim3, 3)
- **energy_type** – type of the energy
- **kwargs** – absorb additional arguments

Returns shape = (batch,)

3.22 Optimizer

Functions parsing the config optimizer options

deepreg.model.optimizer.**build_optimizer**(*optimizer_config: dict*)
Parsing the optimiser options and parameters from config dictionary.

Parameters **optimizer_config** – unpacked dictionary for the optimiser returned from yaml.load, optimiser options and parameters

Returns tf.keras.optimizers object

3.23 Guideline

We welcome contributions to DeepReg, please follow the insutruction for setting up your development environment before editing source code.

- For reporting bugs, or to request features, please raise an issue.
- For fixing bugs, or implementing features, please send a pull request.
- For adding DeepReg Demos, please check the additional requirements in Add a DeepReg Demo.

3.24 Set Up

To edit the source code of DeepReg, besides the package installation, we recommend installing pre-commit for code style consistency and auto formatting before each commit to prevent unnecessary linting failure in Travis-CI.

3.24.1 Pre-commit

We are currently using (by order) the following pre-commit hooks:

- seed-isort-config and isort to format package imports in python files.
- Black to format python files.
- Flake8 to perform python linting check,
- Prettier to format markdown files.

Installation

Pre-commit is installed during the package installation via `pip install -e .`. To activate pre-commit, make sure the git is installed (`sudo apt install git` for linux) and run `pre-commit install` under the root of this repository `DeepReg/`.

Usage

We can use `pre-commit run --all-files` to trigger the hooks manually to format all files before pull request.

Optionally, we can use `git commit --no-verify -m "This is a commit message placeholder."` to skip pre-commit. However, this is not recommended.

Linting conflicts

Sometimes, Black might have conflicts with flake8 and below are some possible cases and work around.

- If a code is followed by a long comment in the same line, Black attempts to break lines. So we should put comment in the line above instead.
- For lists/tuples, do not add comma after the last element, unless it's a single element tuple, like `(1,)`.

To check if Black is causing conflicts, run `black .` in the root of DeepReg you will see the formatted files by Black, run `pre-commit run --all-files`, you will see the final versions. Compare them to understand an issue. If there's a new conflict case, please raise an issue.

3.24.2 Conda Environment

We recommend using `conda env create -f environment.yml` to create the conda environment in installation. In case that we change the dependencies later, please use `conda env update -f environment.yml` to update the packages.

Otherwise, we can always remove the environment using `conda env remove -n deepreg` and recreate it using `conda env create -f environment.yml`.

3.25 Send a Pull Request

We recommend using fork to do pull requests:

1. Fork the repository
2. Create a branch for your changes. The branch name should start with the issue number, followed by hyphen separated words describing the issue, e.g. `1-update-contribution-guidelines`.
3. Make your changes following our guidelines:
    - Coding requirement
    - Unit test requirement
    - Documentation requirement
    - Commit requirement
4. Submit a pull request
5. Pull request will be reviewed and, if necessary, changes will be suggested.

3.25.1 Coding

To ensure the code quality and the consistency, we recommend the following guidelines.

Coding design

1. Please use packages that already have been included. Additional libraries will require a longer review for scrutinizing its necessity.

   In case of adding a new dependency, please make sure all dependencies have been added to `requirements.txt`.

2. Please prevent adding redundant code. Try wrapping the code block into a function or class instead.

   In case of adding new functions, please make sure the corresponding unit tests are added.

3. When adding/modifying functions/classes, please make sure the corresponding docstrings are added/updated.
4. Please check Unit test requirement for detailed requirements on unit testing.
5. Please check Documentation requirement for detailed requirements on documentation,

3.25.2 Testing

In DeepReg, we use pytest (not unittest) for unit tests to ensure a certain code quality and to facilitate the code maintenance.

The testing is checked via Travis-CI and Codecov is used to monitor the test coverage. While checking the Codecov report in file mode, generally a line highlighted by red means it is not covered by test. In other words, this line has never been executed during tests. Please check the Codecov documentation for more details about their coverage report.

Following are the guidelines of test writing to ensure a consistency of code style.

Coverage requirement

Class

For a class, we need to test all the functions in the class.

Non-TensorFlow function

For a non-TensorFlow function, we need to test

- The correctness of inputs and the error handling for unexpected inputs.
- The correctness of outputs given certain inputs.
- The trigger of all errors (`ValueError`, `AssertionError`, etc.).
- The trigger of warnings.

Check `test_load_nifti_file()` in test_nifti_loader.py as an example.

TensorFlow function

For a TensorFlow-involved function, we need to test

- The correctness of inputs and the error handling for unexpected inputs. The minimum requirement is to check the shape of input tensors.
- The correctness of outputs given certain inputs if the function involves mathematical operations. Otherwise, at least the output tensor shapes have to be correct.
- The trigger of all errors (`ValueError`, `AssertionError`, etc.).
- The trigger of warnings.

Check `test_resample()` in test_layer_util.py as an example.

Helper functions

As we are comparing often the numpy arrays and TensorFlow tensors, two functions `is_equal_np` and `is_equal_tf` are provided in test/unit/util.py. They will first convert inputs to float32 and compare the max of absolute difference with a threshold at 1e-6. They can be imported using `from test.unit.util import is_equal_np` so that we do not need one copy per test file.

Example unit test

In this section, we provide some minimum examples to help the understanding.

Function to be tested

Assuming we have the following function to be tested:

```
import logging

def subtract(x: int) -> int:
    """
    A function subtracts one from a non-negative integer.
    :param x: a non-negative integer
    :return: x - 1
    """
    if not isinstance(x, int):
        raise ValueError(f"input {x} is not int")
    assert x >= 0, f"input {x} is negative"
    if x == 0:
        logging.warning("input is zero")
    return x - 1
```

Coverage requirement

The test should be as follows:

- Name should be the tested function/class with prefix `test_`, in this example, it is `test_subtract`.
- All cases are separated by a comment briefly explaining the test case.
- Test a working case, e.g. input is 0 and 1.
- Test a failing case and the `assert`, e.g. input is -1. We need to catch the error and check the error message if existed.
- Test the `ValueError`, e.g. input is 0.0, a float. We need to catch the error and check the error message if existed.
- Verify the warning is trigger, e.g. input is 0.

Test code

```
import pytest

def test_subtract(caplog):
    """test subtract by verifying its input and outputs"""
    # x = 0
    got = subtract(x=0)
    expected = -1
    assert got == expected

    # x > 0
    got = subtract(x=1)
    expected = 0
    assert got == expected

    # x < 0
    with pytest.raises(AssertionError) as err_info:
        subtract(x=-1)
    assert "is negative" in str(err_info.value)

    # x is not int
    with pytest.raises(ValueError) as err_info:
        subtract(x=0.0)
    assert "is not int" in str(err_info.value)

    # detect caplog
    caplog.clear() # clear previous log
    subtract(x=0)
    assert "input is zero" in caplog.text

    # incorrect warning test example
    # caplog.clear()  # uncomment this line will fail the test
    subtract(x=1)  # this line generates no warning
    assert "input is zero" in caplog.text

    # incorrect error test example
    with pytest.raises(AssertionError) as err_info:
        # this line will trigger the Assertion Error
        # comment the following line will fail the test
```

(continues on next page)

(continued from previous page)

```
        subtract(x=-1)
        # the following line will never be executed
        subtract(x=0.0)
    assert "is negative" in str(err_info.value)
```

Common errors

Forget to clear caplog

In the example above, we are testing warning messages with pytest caplog fixture. All the messages are captured in `caplog` which is the input argument of the test function. Be careful that it is important to clear the caplog using `caplog.clear`. Otherwise, as the log is accumulated, we might have unexpected performance.

For instance, with the example above:

```
# incorrect warning test example
# caplog.clear()  # uncomment this line will fail the test
subtract(x=1)  # this line generates no warning
assert "input is zero" in caplog.text
```

The test will pass but the assertion works only because we have generated. If the `caplog.clear()` is uncommented, the test will fail.

Test multiple errors together

When testing errors, the `assert` should be outside of the `with pytest.raises(ValueError) as err_info:` and we should not put multiple tests inside the same `with` as only the first error will be captured.

For instance, in the following test example, the second subtract will never be executed regardless of whether it is correct or not. If this trigger an error, it will never be captured.

```
# incorrect error test example
with pytest.raises(AssertionError) as err_info:
    # this line will trigger the Assertion Error
    # comment the following line will fail the test
    subtract(x=-1)
    # the following line will never be executed
    subtract(x=0.0)
assert "is negative" in str(err_info.value)
```

The test will pass because the first subtract raises the desired assertion error. The test will fail if we comment out the first subtract.

3.25.3 Documentation

We use Sphinx to organize the documentation and it is hosted in ReadTheDocs.

Build

Please run the following command under `docs/` directory for generating the documentation pages locally. This is also included in Travis-CI jobs to ensure the documentation quality.

```
make clean html
```

where

- `clean` removes the possible built files.
- Optionally we can add `SPHINXOPTS="-W"` to prevent warnings, but we are currently having `document isn't included in any toctree` warning and no better solution has been found yet.

Recommendations

There are some recommendations regarding the docs.

- **We prefer markdown files** over reStructuredText files as its linting is covered using Prettier.

  Only use reStructuredText (rst) files for some functionalities not supported by markdown, such as

  - `toctree`
  - warning/notes boxes in Installation

  The conversion between markdown and rst can be done automatically using free online tool Pandoc.

- When linking to other pages, **please use relative paths** such as `../getting_started/install.html` instead of absolute paths `https://deepreg.readthedocs.io/en/latest/getting_started/install.html` as relative paths are more robust for different version of documentations.
- To refer a markdown file outside of the source folder, create an rst file and use `.. mdinclude:: <makrdown file path>` to include the markdown source.

  Check the source code of paired lung CT image registration page as an example.

3.25.4 Commit

To facilitate review of contributions, we encourage contributors to adhere to the following commit guidelines.

1. The commit message should start with `Issue #<issue number>:` so that the commits are tied to a specific issue.

   - **Good**:

     ```
     Issue #1: modified resample function docstring to reflect changes in function␣
     ↪args
     ```

   - **Moderate/OK**:

     Inconsistent commit style.

     ```
     ref #<issue number>: modified resample function docstring to reflect changes␣
     ↪in function args
     ```

   - **Bad**:

     Missing issue number.

```
modified resample function docstring to reflect changes in function args
```

2) Include related changes in the same commit, where possible, For example, if multiple typos are spotted in a document, aim to resolve them in the same commit instead of separate commits. This improves history readability.

- **Good**:

  One commit for the same type of problem

```
Issue #<issue number>: fixed typos in function x
```

- **Bad**:

  Multiple commits of the same message in the same thread. Clutters repo history.

3) Strive to add informative commit messages.

- **Good**:

```
Issue #<issue number>: removed unused arguments across function x in loops to␣
↪comply with PEP8 standard in file y
Issue #<issue number>: added new data loader class inheriting from base class to␣
↪deal with np array file format
```

- **Not acceptable**:

  Not enough details, hard to tell explicitly what was changed without doing an in depth review.

```
Issue #<issue number>: lint
Issue #<issue number>: add loader
```

3.26 Add a DeepReg Demo

The demos folder directly under the DeepReg root directory contains demonstrations using DeepReg for different image registration applications.

Contributions are welcome! Below is a set of requirements for a demo to be included as a DeepReg Demo.

- Each demo *must* have an independent folder directly under `demos/`;
- Name the folder as `[loader-type]_[image-modality]_[organ-disease]_[optional:brief-remark]`, e.g. `unpaired_ultrasound_prostate` or `grouped_mr_brain_longitudinal`;
- For simplicity, avoid sub-folders (other than those specified below) and separate files for additional functions/classes;
- Experiment using cross-validation or advanced data set sampling is NOT encouraged, unless the purpose of the demo is to demonstrate how to design experiments.

3.26.1 Open accessible data

- Each demo *must* have a `demo_data.py` script to automatically download and preprocess demo data;
- Data should be downloaded under the demo folder named `dataset`;
- Data should be hosted in a reliable and efficient (DeepReg repo will not store demo data or model) online storage, Kaggle, GitHub and Zendoo are all options for non-login access (avoid google drive for known accessibility issues);
- Relevant dataset folder structure to utilise the supported loaders can be either pre-arranged in data source or scripted in `demo_data.py` after downloading;
- Avoid slow and excessively large data set download. Consider downloading a subset as default for demonstration purpose, with options for full data set.

3.26.2 Pre-trained model

- A pre-trained model *must* be available for downloading, with github.com/DeepRegNet/deepreg-model-zoo being preferred for storing the models. Please contact the Development Team for access;
- The pre-trained model, e.g. ckpt files, should be downloaded and extracted under the `dataset` folder. Avoid overwriting with user-trained models;

3.26.3 Training

- Each demo *must* have a `demo_train.py` script. If using command line interface `deepreg_train`, this file needs to print a message to direct the user to the readme.md file (described below) for instructions;
- This is accompanied by one or more config yaml files in the same folder. If appropriate, please use the same demo folder name for the config file. Add postfix if using multiple config files, e.g. `unpaired_lung_ct_dataset.yaml`, `unpaired_lung_ct_train.yaml`.

3.26.4 Predicting

- Each demo *must* have a `demo_predict.py` script; If using command line interface `deepreg_predict`, this file needs to print a message to direct users to the readme.md file (described below) for instructions;
- By default, the pre-trained model should be used in `demo_predict.py`. However, the instruction should be clearly given to use the user-trained model, saved with the `demo_train.py`;
- Report registration results. Provide at least one piece of numerical metric (Dice, distance error, etc) to show the efficacy of the registration. Optimum performance is not required;
- Provide at least one piece of visualisation of the results, such as moving image vs fixed image vs warped moving image (pred_fixed_image). This may be simply done by selecting the typical results from the predict output. If possible, save the visualisation to (e.g. png/jpg) files, avoiding compatibility issues. Pointing to the relevant file paths generated using `deepreg_predict` is recommended.

3.26.5 A README.md file

The markdown file *must* be provided as an entry point for each demo, which should be based on the template.

Following is a checklist for modifying the README template:

- Modify the link to source code;
- Modify the author section;
- Modify the application section;
- Modify the data section;
- Modify the steps in instruction section;
- Modify the pre-trained model section;
- Modify the tested version;
- Modify the reference section.

3.27 Packaging a Release

`DeepReg` is disributed on PyPI. To create new releases, you can follow the below instructions and submit new versions to PyPI.

3.27.1 Prerequisites

Make sure you have `setuptools`, `wheel`, and `twine` installed in your environment:

```
pip install setuptools wheel twine
```

Update your `setup.py` file with the appropriate version number. Then, from within the `DeepReg` folder where you want the version to be built from:

```
python setup.py sdist bdist_wheel
```

3.27.2 Upload to TestPyPI

Once built, upload to `testpypi` to ensure that the package runs as expected prior to adding the new release to `pypi`. This will require an account on `testpypi`.

```
twine upload --repository testpypi dist/*
```

You'll be prompted to enter your username and password.

3.27.3 Upload to PyPI

If all works well, upload to `pypi`:

```
twine upload dist/*
```

You'll, once again, be prompted to enter your username and password.

3.27.4 Tag & upload the release to GitHub

Make sure to add the release to the DeepReg repository on GitHub as well. Make sure to follow the given naming conventions for tags with `vX.Y.Z` with major, minor, and batch releases.

PYTHON MODULE INDEX

d

INDEX

D

F

G

H

I

J

L

M

N

P

R

S

T

U

V

W