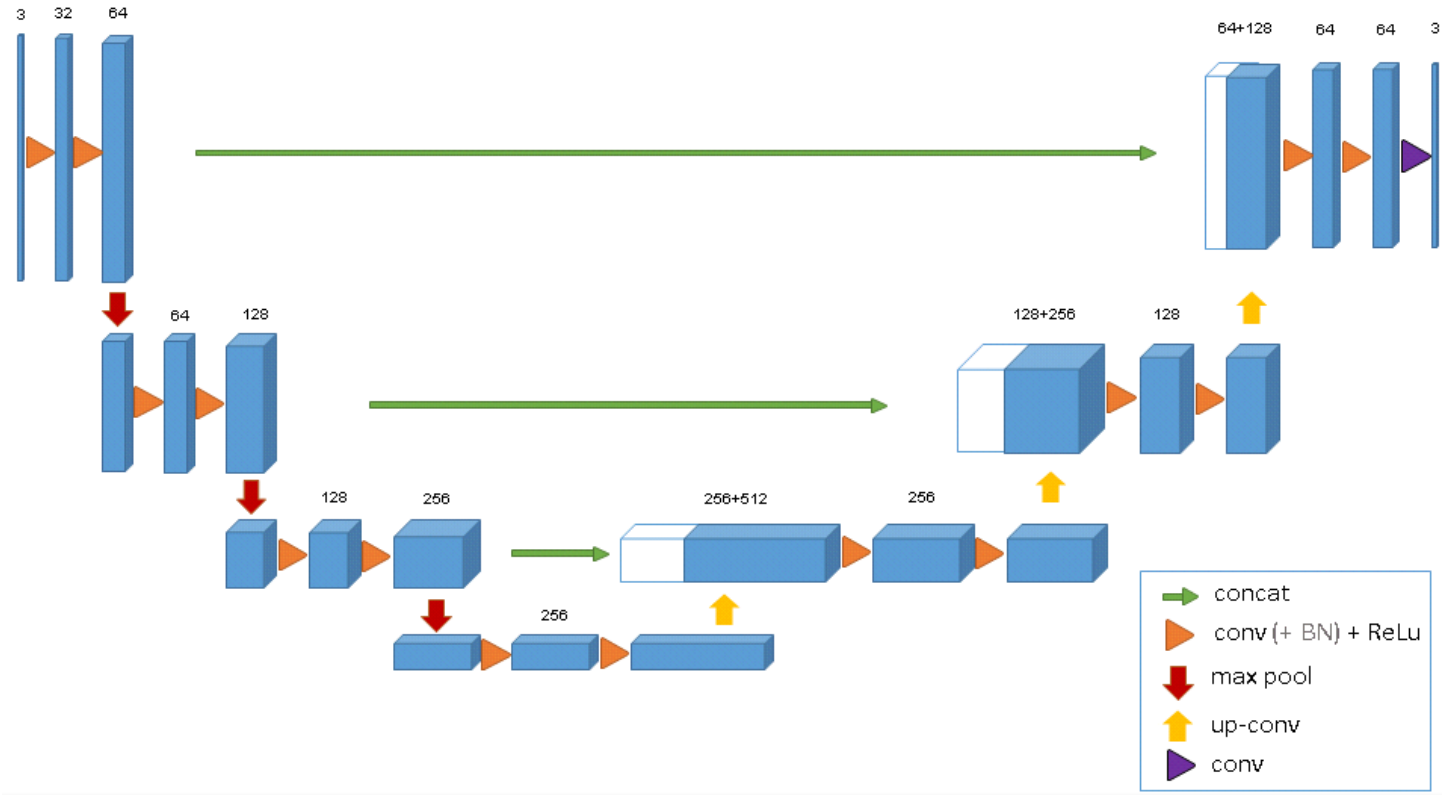
3D U-Net Segmentation

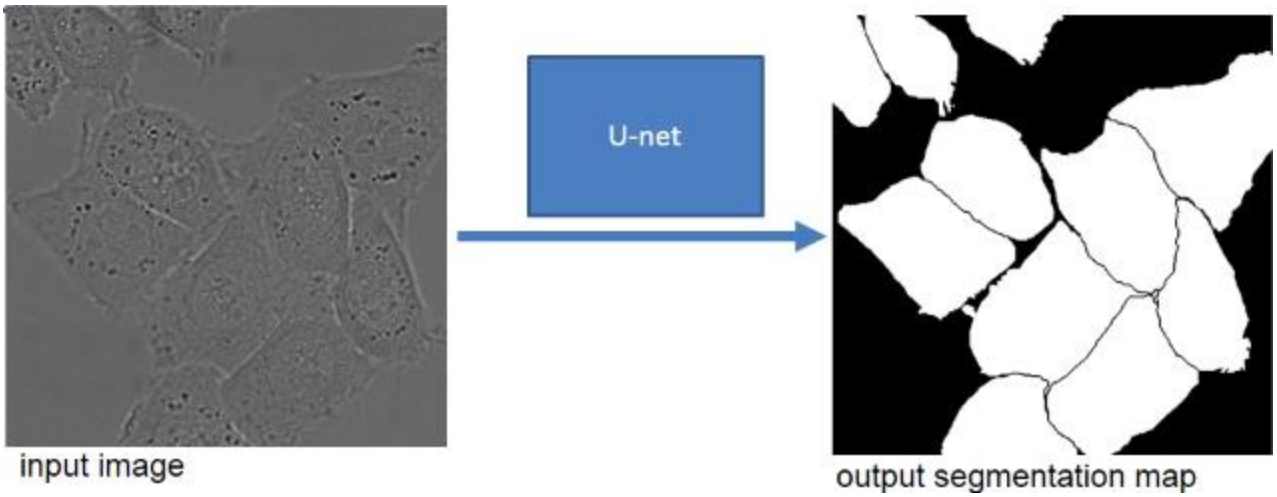
Abstract

As a part of a deep convolutional neural network, the 3D U-Net segmentation introduces a network and training strategy that is based on the usage of data augmentation to the available samples more efficient. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization.

What is 3D U-Net Segmentation?

3D U-Net segmentation is an architecture based on the Convolutional Neural Network (CNN), which has typical use to classify labels. However, in medical imaging, the desired output should be more than just classification. It should contain the localization that is set up to predict the class label of each pixel by providing a local region around that pixel as an input.





Dataset

In this experiment, we use the dataset BraTS 2017, the dataset for brain tumors. The differentiation between low-grade gliomas (LGGs; grade II) and high-grade gliomas (HGGs; grades III, IV) is critical, since the prognosis and thus the therapeutic strategy could differ substantially depending on the grade.

Glioblastoma (GBM) is a highly aggressive (grade 4) type of brain tumor, with cells that reproduce quickly—nourished by a large network of blood vessels. GBM is the most common type of HGG, with an annual incidence of approximately 13,000 in 2019 in the United States. It accounts for about half of all primary brain and central nervous system cancers.

The architecture

Originally designed after this paper on volumetric segmentation with a 3D U-Net. U-Net uses a weighted cross-entropy as its loss function. The per-pixel weights are given by a formula which:

- Balances the weights between classes and
- Has an extra term to penalize joining two bits of the segmentation.

```
def unet_model_3d(input_shape, pool_size=(2, 2, 2), n_labels=1, initial_learning_rate=0.00001, deconvolution=False,
                  depth=4, n_base_filters=32, include_label_wise_dice_coefficients=False, metrics=dice_coefficient,
                  batch_normalization=False, activation_name="sigmoid"):
```

With some default values are set in the 3D U-Net model initiation: pool size, the number of labels, the initial learning rate, no deconvolution, the depth is four, the number of base filters is thirty-two, label wise dice coefficients provided, activation is sigmoid, etc.

Preprocessing the Data

We first download the source code from here and unzip the dataset to the folder brats/data/original. There are some dependencies need to install such as nibabel, keras, pytables, nilearn, SimpleITK, nipype. Unfortunately, the python 3.x does not support the nipype well because the build currently has some test cases failed as pictures shown below.

We download the ANTs (should provide the link to the other document) from here (the version should be 2.3.1) and prepare to configure the environment to preprocess the dataset. We also need to run cmake to build the dependency from the source code then run configure command to generate default values for the installation.

There are some scripts executed from the python code, so we need to add the repository directory of ANTs N4BiasFieldCorrection into the PYTHONPATH system variable by navigating to the script folder and run this command line export PYTHONPATH=${PWD}:$PYTHONPATH

We now convert the data to NIFTI format and perform image-wise normalization and correction. First, we get in the folder brats and run the python command line from preprocessing import convert_brats_data to import the package and then convert_brats_data("data/original", "data/preprocessed") to normalize and correct the input. On a single GPU, it should take up to twenty minutes.

```
def correct_bias(in_file, out_file, image_type=sitk.sitkFloat64):
    """
    Corrects the bias using ANTs N4BiasFieldCorrection. If this fails, will then attempt to correct bias using SimpleITK
    :param in_file: input file path
    :param out_file: output file path
    :return: file path to the bias corrected image
    """
    correct = N4BiasFieldCorrection()
    correct.inputs.input_image = in_file
    correct.inputs.output_image = out_file
    try:
        done = correct.run()
        return done.outputs.output_image
    except IOError:
        warnings.warn(RuntimeWarning("ANTs N4BIasFieldCorrection could not be found."
                                     "Will try using SimpleITK for bias field correction"
                                     " which will take much longer. To fix this problem, add N4BiasFieldCorrection"
                                     " to your PATH system variable. (example: EXPORT PATH=${PATH}:/path/to/ants/bin)"))
        input_image = sitk.ReadImage(in_file, image_type)
        output_image = sitk.N4BiasFieldCorrection(input_image, input_image > 0)
        sitk.WriteImage(output_image, out_file)
        return os.path.abspath(out_file)
```

The process basically normalizes the image input and correct the bias and write out a new image output.

Data Generator

We can also define the generator function for the model which has some features that hold in the x list and the class label (ground truth) in the y list. A list in here is usually a numpy array, and it needs to be converted.

```
def data_generator(data_file, index_list, batch_size=1, n_labels=1, labels=None,
                   augment=False, augment_flip=True, augment_distortion_factor=0.25,
                   patch_shape=None, patch_overlap=0, patch_start_offset=None,
                   shuffle_index_list=True, skip_blank=True, permute=False):
    orig_index_list = index_list
    while True:
        x_list = list()
        y_list = list()
        if patch_shape:
            index_list = create_patch_index_list(orig_index_list, data_file.root.data.shape[-3:],
                                                 patch_shape, patch_overlap, patch_start_offset)
        else:
            index_list = copy.copy(orig_index_list)
        if shuffle_index_list:
            shuffle(index_list)
        while len(index_list) > 0:
            index = index_list.pop()
            add_data(x_list, y_list, data_file, index, augment=augment, augment_flip=augment_flip,
                     augment_distortion_factor=augment_distortion_factor, patch_shape=patch_shape,
                     skip_blank=skip_blank, permute=permute)
            if len(x_list) == batch_size or (len(index_list) == 0 and len(x_list) > 0):
                yield convert_data(x_list, y_list, n_labels=n_labels, labels=labels)
                x_list = list()
                y_list = list()

def convert_data(x_list, y_list, n_labels=1, labels=None):
    x = np.asarray(x_list)
    y = np.asarray(y_list)
    if n_labels == 1:
        y[y > 0] = 1
    elif n_labels > 1:
        y = get_multi_class_labels(y, n_labels=n_labels, labels=labels)
    return x, y
```

Training the Data

We use original U-Net model here to train the data by running the command python train.py. The training process should take up to three or four hours. You may need to adjust the path between the folders to make sure it executes in the right place. You should have four output files such as brats_data.h5, tumor_segmentation_model.h5 if the training is successful.

```
config = dict()
config["pool_size"] = (2, 2, 2)  # pool size for the max pooling operations
config["image_shape"] = (144, 144, 144)  # This determines what shape the images will be cropped/resampled to.
config["patch_shape"] = (64, 64, 64)  # switch to None to train on the whole image
config["labels"] = (1, 2, 4)  # the label numbers on the input image
config["n_labels"] = len(config["labels"])
config["all_modalities"] = ["t1", "t1ce", "flair", "t2"]
config["training_modalities"] = config["all_modalities"]  # change this if you want to only use some of the modalities
config["nb_channels"] = len(config["training_modalities"])
if "patch_shape" in config and config["patch_shape"] is not None:
    config["input_shape"] = tuple([config["nb_channels"]] + list(config["patch_shape"]))
else:
    config["input_shape"] = tuple([config["nb_channels"]] + list(config["image_shape"]))
config["truth_channel"] = config["nb_channels"]
config["deconvolution"] = True  # if False, will use upsampling instead of deconvolution
```

Before training the data, we need to reconfigure the information above to fit the experiment's purpose. The pool size must be at least two for each dimension. The labels represent the class labels of the medical imaging such as a whole tumor, enhanced tumor, etc. The modalities have four types by default, we can take off some for a specific purpose.

```
def fetch_training_data_files():
    training_data_files = list()
    for subject_dir in glob.glob(os.path.join(os.path.dirname(__file__), "data", "preprocessed", "*", "*")):
        subject_files = list()
        for modality in config["training_modalities"] + ["truth"]:
            subject_files.append(os.path.join(subject_dir, modality + ".nii.gz"))
        training_data_files.append(tuple(subject_files))
    return training_data_files
```

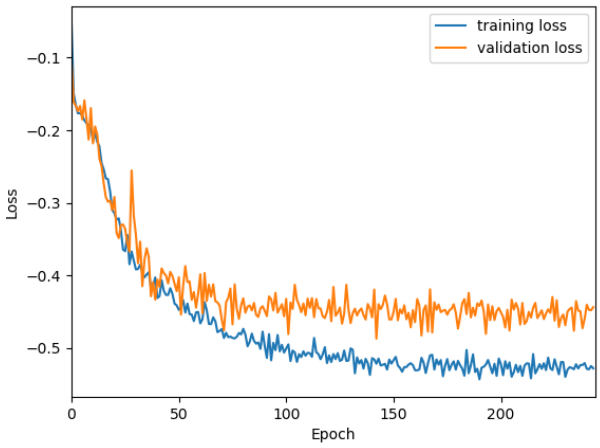
The function fetches all training data files to separate the training modalities in the subject list.

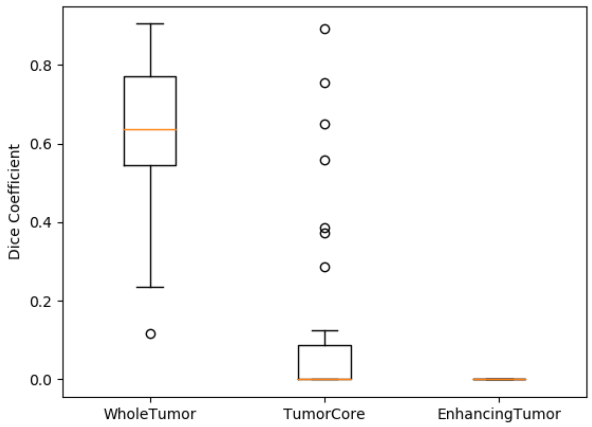
Write the prediction images

From the same dataset, a part of data was held for validation purposes, and now we are going to write the predicted label maps to file training_ids.pkl and validation_ids.pkl. The predictions will be written in the folder prediction along with the input data and ground truth labels for comparison. It should take up to twenty minutes.

Evaluate the model

After running the script evalutate.py in the brats folder, you should have both the loss graph and the box plot.





References

[1] https://github.com/ellisdg/3DUnetCNN

[2] https://github.com/ANTsX/ANTs

[3] https://github.com/nipy/nipype

[4] https://www.med.upenn.edu/sbia/brats2017.html

[5] https://lmb.informatik.uni-freiburg.de/Publications/2016/CABR16/cicek16miccai.pdf