Badblock Documentation

Release 1.0

Liang Li

May 30, 2019

HERE ALL THE CONTENTS GO:

Deep learning has been fast developed in recent years, which triggers lots of researches in medical image analysis. In this tutorial, I will show you how to improve the quality of image(or say image denoising/impainting) by using one of the many networks, UNet.

CHAPTER

ONE

REASONS FOR THIS TUTORIAL

I started my internship in Siemens, Knoxville at the begining of 2019, where I first touched and learned deep learning. Great thanks to super nice manager(BILL) since I learn a lot from him!

During my internship, I try to use deep learning(UNet, FrameletNet, GAN) to do image impainting or denoising, while I think this is so great a chance for me to learn about deep learning and explore different kinds of networks, and this makes me feel excited almost every day so I want to share this with you.

In fact, my coworkers are also very interested in building or learning neural networks while their think that deep learning is very hard stops them. And thus, I also mean to write this tutorial for people who are in the field of medical imaging who might want to build their own deep learning networks but they do not know how/where to start.

To be honest, in either deep learning or machine learning, the optimal choices for the most suitable parameters/models are always hard to make and it is also consuming most of the time, but initiating and building deep learning network and making it work is far more simple than you thought!

Last but not least, I am still a beginner and learner in deep learning/matchine learning, and please point/teach me out if I will be wrong in the following content.

CHAPTER

TWO

PIPELINE FOR THIS TUTORIAL

Here, let's begin with the following content to open your new world to the deep learning(AI)!

First, we will install all the necessary packages/softwares for the deep learning(it will be great if you have a GPU), and in this tutorial we will utilize Pytorch.

Next, we will talk about how to precesss your data (so important), such as shaping and normalization! And I will always try to add a method named augumator to further enlarge our datasets in case that the training data is very small.

Third, we will go over neural network(U-Net), which will let you learn the core of deep learning. In this part, I will briefly introduce the idea and the structure of U-Net.

Finally, we will train our U-Net with our training datasets and use visualization tool to view our results.

2.1 Install necessary packages

Python is a programming language that lets you work quickly and integrate systems more effectively. From my point, python is simple to use and learn. The reason why we use python is that currently most deep learning frameworks have already been implemented based on python and plenties of open source packages that are available in the field of data science can be utilitized for our data analysis, such as scipy, scikit-learn, pandas. Pythonic makes life easier.

2.1.1 Install Python

Direct install Python from *Python*

Python2 will not be supported any more by the community, and hence let's work on Python3 and install the latest python. The latest Python can be downloaded and installed from https://www.python.org/downloads/ . I have installed 3.6, please try to install a version above 3.6.

Indirect install Python from *Conda*

Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies.

Conda can be found via https://docs.conda.io/projects/conda/en/latest/user-guide/install/.

Conda is recommended when multiple versions of python will be installed in the same system, and the version of python can be changed easily by using:

```
source activate myenv
```

For example,

```
source activate py36
```

with the whole list of pythons with different versions can be shown as:

```
conda env list
```

other useful methods for install certain packages including

```
conda search scipy
conda install --name myenv scipy
conda install scipy=0.15.0
source deactivate
conda info --envs
conda list -n myenv scipy
```

2.1.2 Install Pytorch with CUDA

Before we start with pytorch, please make sure CUDA has been installed, where CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.

Install CUDA Driver

Download and install your cuda driver from https://developer.nvidia.com/cuda-downloads?target_os=Linux.

Note: Please check the same link for updating with the latest driver. A good match of driver with the GPU will largerly increase the speed, so always make sure you have the latest driver with your GPU.

Install pytorch

Pytorch is an open source deep learning platform that provides a seamless path from research prototyping to production deployment.

pytorch can be found via https://pytorch.org/get-started/locally/.

2.1.3 Other packages

There are a list of other packages that are optional to install, while most could be install by using **pip**. [if you are under conda, make sure you are using the correct **pip** under the correct version of python]

- $ pip install visdom
- $ pip install numpy
- $ pip install matplotlib
- $ pip install Pillow
- $ pip install scipy
- $ pip install Augmentor

Plus, if you would consider mssim loss too, please include 'pytorch_msssim' folder{we will talk about this later} and if there are other package needed, try **pip**

```
pip install packagename
```

NOW, you are all set with all the packages needed for the deep learning Unet, and in the next step we will forward to prepare our data.

2.1.4 Build in Docker (Optional)

If GPU or the practical hardware is unavailable, we can also learn the deep learning by utilizing all the cloud services availble, such as AWS. Here, a simple system Docker is covered for the cases where we need to run deep learning in a server.

Docker is a computer program that performs operating-system-level virtualization. The advantage of Docker is that we can only install the packages we need and then the extra cost of unnecessary components in the operating system can be reduced.

See also:

If a python environment with pytorch is hard to obtain locally, Docker is always a good choice to make your network run in cloud. Note: a most recent pytorch with NVIDIA can be pulled from https://docs.nvidia.com/deeplearning/dgx/pytorch-release-notes/running.html.

Docker is simple to use too! The followings are a summary of Docker codes for reference.

```
# List Docker images
     docker image ls

# List Docker containers (running, all, all in quiet mode)
     docker container ls
     docker container ls --all
     docker container ls -aq
     docker container stop xxxxxxx
# List Docker containers (running, all, all in quiet mode)
     docker build -t friendlyhello .  # Create image using this directory's Dockerfile
     docker run -p 4000:80 friendlyhello  # Run "friendlyname" mapping port 4000 to 80
     docker run -d -p 4000:80 friendlyhello         # Same thing, but in detached mode
     docker container ls                                # List all running containers
     docker container ls -a             # List all containers, even those not running
     docker container stop <hash>           # Gracefully stop the specified container
     docker container kill <hash>         # Force shutdown of the specified container
     docker container rm <hash>        # Remove specified container from this machine
     docker container rm $(docker container ls -a -q)         # Remove all containers
     docker image ls -a                             # List all images on this machine
     docker image rm <image id>            # Remove specified image from this machine
     docker image rm $(docker image ls -a -q)   # Remove all images from this machine
     docker login             # Log in this CLI session using your Docker credentials
     docker tag <image> username/repository:tag  # Tag <image> for upload to registry
     docker push username/repository:tag            # Upload tagged image to registry
     docker run username/repository:tag                   # Run image from a registry
```

Using Docker within DGX

Special Requirement for DGX user: In order to connect to his Docker daemon a user has to commit the parameter "-H unix:///mnt/docker_socks/<user_name>/docker.sock" with every Docker command.

- e.g. "docker -H unix:///mnt/docker_socks/<user_name>/docker.sock run –rm -ti <image_name> [optional_command]"
- e.g. "docker -H unix:///mnt/docker_socks/<user_name>/docker.sock image ls" alternatively use the script "run-docker.sh" in /usr/local/bin:
- e.g. "run-docker.sh –rm -ti [further_options] <image_name> [optional_command]"

I was using a line like below in a bash file to let the docker run within DGX by using slum.

```
sbatch --gres=gpu:1 --mail-user=liang.li.uestc@gmail.com --mail-type=ALL --output=/
↪badblock/li_dataset/log.txt --job-name=trainbd --error=/badblock/li_dataset/error.
↪txt run-nvidia-docker.sh --rm --name train_unet -v /badblock/li_dataset/:/badblock/␣
↪-w /badblock/ nvcr.io/nvidia/pytorch:19.01-py3 python /badblock/code/badblock-
↪fillgaps/train_unet.py --training-file="/badblock/data/DataTOF_Train/" --test-file=
↪"/badblock/data/DataTOF_test/" --mask-file="/badblock/data/mask.pkl" --output-path=
↪"/badblock/output/"
```

2.2 Prepare your data

The size(shape) and the distribution of the data would affect both the performance and the learning speed of the network, and hence reshaping or preprocessing the raw data to the shape/distribution we want and post-processing it back to the origin format are usually common in machine learning [The reasons behind this are a lot, say we want the learning converge similar to all directions in our training data]. In most the cases, the methods include but not limit to normalization, reshaping, etc.

2.2.1 Know our raw data

Before we prepare/process our data, it would be better if we know what data we are going to train/learn. For example, for images, we can view the images by using:

```
plt.imshow(image)
```

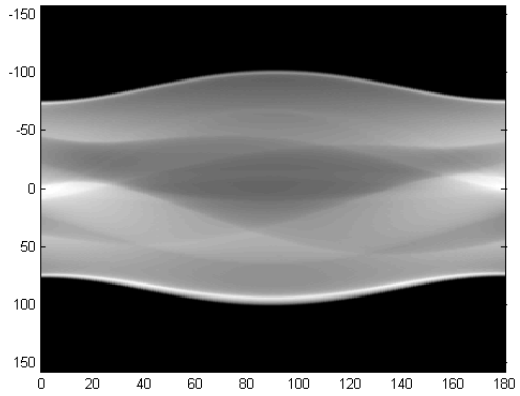
where plt is short for matplotlib.pylab. In this way, we can have a roughly idea about our dataset/image.

In the following, I give two examples about the data we deal with for medical imaging, sinograms and images.

eg. Sinograms

PET-CT and PET-MR scanners store the raw data in proprietary formats which can be processed only by the software provided by the scanner manufacturer, where one of the raw data is sinogram, which is basically 2D representation of projection rays versus angle. And different from RGB images, the pixels in sinograms stand for the counts of events captured from the scanner, which range from zero to thousands or tens of thousands.

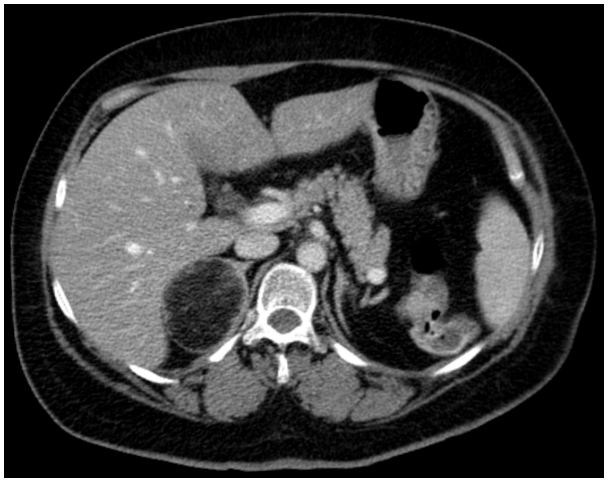
One sample of sinogram is shown below.



If our goal is to fix the noise in sinograms of one patient, say, we have the sinograms with the format as .s with the input sinograms with noise and the target sinograms without noise. Then, we can use U-Net to project the data from the input files and to make it close to the data in the target file (ground true).

Actually, the practical sinogram file of one patient can be very big, around several GBs, and hence the best way to train is not to feed all the sinograms into memory but to seperate the single sinograms of one patient into small parts, which would be more beneficial to reduce the cost of both memory and learning speed.

eg. Images



To clarify, our final goal can be to fix the noise in body image of one patient (to make the image more clear and help the doctors make good decisions). Thus, rather than working on singrams, another direct solution will be to work on the body images directly. For instance, we feed the reconstructed image with noise into our neural networks as inputs and our target will be the images without noise.

Note, the above noise can also be interperated as impainting.

2.2.2 Process our raw data

After getting an idea about our dataset, now we can proceed in processing the raw dataset.

Partition

As stated, to advoid loading all data into the memory, we would better reshape our dataset into more informative matrix and partition the dataset into smaller pieces. In a large scale, the memory consumption will be reduced and the learning speed will be accelerated. However, we might also lose the relations/connections among the smaller pieces.

For an example, I am working on sinograms and the sinograms have their own informative structure [TOF, Slice, W, L], where TOF stands Time of Flight, Slice stands the slices of the sinogram, W, L stand for the rays versus angle. After reshaping the dataset from [TOF, Slice, W, L] into [TOF*Slice, W, L], I feed the network with the sinogram based on slice. For example, for each slice, the sinogram has the shape [W, L]. In this way, we could definitely reduce the cost of memory, but we also lose the correlative information among slices.

Padding

For most cases in machine learning without down/up sampling in the images, the number of W and L does not matter. Since we are working on UNet (as autoencoder, we need to encode and decode the data) we would better make W and L as a power of 2 (since we will consider the network structure as UNet which will include both downsampling and upsampling).

In this case, if W and L is close to certain number which is a power of 2, and then we can match W and L to the numbers by using padding in numpy.

Note: An example of reshaping to a power of 2:

Numpy provides the padding function:

```
data = np.pad(data, ((x1, y1), (x2, y2), (x3, y3), (x4, y4)), 'wrap')
```

For example, if we have [TOF=1, Slice=1, W=50, L=256], we can do:

```
data = np.pad(data, ((0, 0), (0, 0), (7, 7), (0, 0)), 'wrap')
```

to make it [TOF=1, Slice=1, W=64, L=256], which would be good enough for 3 times downsampling since W and L can be divided by 2^3.

Normalization

Normalization is usualy called to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. And there are plenties of methods which can be utilized and explored. Here I proposed the most easy one by mapping the dataset to [0,1].

The code can be easily in Python as:

```
data = data / (data.max() + 1e-8)
```

The small value 1e-8 is necessary, especially when the values of the images are integer. In this way, we can map the value to float while between 0 and 1.

Here I only list one way to change the shape, and actually there might be tons of other methods which are good to try for the data processing.

2.2.3 Save the data in pickle

In order to make dataset read more efficiently by python, we save the reshaped dataset into pickle as the intermedium files for the datasets. Pickle module implement binary protocols for serializing and de-serializing a Python object structure.

e.g. we can dump our reshaped matrix directly into pickle by using:

```
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Note: Be careful with you pickle version, since it might not match between python2 and python3.

One example about how to process the dataset is summarized as:

```
def process_data(self, file, tof, slices, theta, dist):

    data = np.fromfile(file, dtype='uint16')
    data = data.reshape((tof, slices, theta, dist))  # Reshape to the file layout

    data = np.pad(data, ((0, 0), (7, 7), (0, 0)), 'wrap')

    return data

result = process_data(file, tof, slices, theta, dist)

with open("savefile.pkl", 'wb') as f:
    pickle.dump(result, f, pickle.HIGHEST_PROTOCOL)
```

The details of the data process can be refered from sino_process_tof.py

Warning: Loading all the data into the GPU is both time consuming and inefficient and hene balancing the between the size before training.

2.2.4 Load your data in batch

Load from dataset class in Pytorch

Before we start, let's see how pytorch works with dataset. Pytorch a Python-based scientific computing package targeted at two sets of audiences:

- A replacement for NumPy to use the power of GPUs
- a deep learning research platform that provides maximum flexibility and speed

In fact, pytorch has listed very good tutorial for beginners, so I will omit this part here while you can find easily online.

Links to Pytorch Tutorial

PyTorch provides many tools to make data loading easy and hopefully, to make your code more readable. For instance, the abstract class in pytorch **torch.utils.data.Dataset** is the main class to call for loading data, and mainly two methods would be called.

- __len__: which returns the size/length of the dataset
- __getitem__: which returns one sample of the dataset based on the index, such that dataset[i] for index i

In our case, we can define the class `class Sino_Dataset(dataset)` inherits from `torch.utils.data.Dataset`

Here, the length of the dataset is:

```
def __len__(self):

    return self.epoch_size
```

and the item of the dataset is:

```
def __getitem__(self, idx):

     return self.data[idx]
```

the `self.data.shape=[tof*slice, W, L]`

The details of getting item also include shuffling and file updating, which can be viewed as different methods to improve the randomness of the data set.

Augument the data randomly while loading

I have been told that the data is very expensive, so if we do not have enough data for training, what should we do?

For images, we can do image augmentation to expand the datsets. There are lot of ways to do the augmentation, such as flipping, zooming and so on. I have found one package from github https://github.com/mdbloice/Augmentor , which can be utilized easily as:

```
images = [[image for image in corrupt_sino] + [orig_sino[0]]]
                p = Augmentor.DataPipeline(images)
                # p.rotate(1, max_left_rotation=5, max_right_rotation=5)
                # p.flip_top_bottom(0.5)
                # p.zoom_random(1, percentage_area=0.5)
                p.rotate(probability=0.7, max_left_rotation=10, max_right_rotation=10)
                p.zoom(probability=0.5, min_factor=1.1, max_factor=1.5)
                augmented_images = p.sample(1)
                corrupt_sino = np.stack([augmented_images[0][i] for i in range(5)])
                orig_sino = np.stack([augmented_images[0][i] for i in range(5, 6)])
```

In this way, the images have been augumented. Other methods are also welcome!

2.3 Build your network!

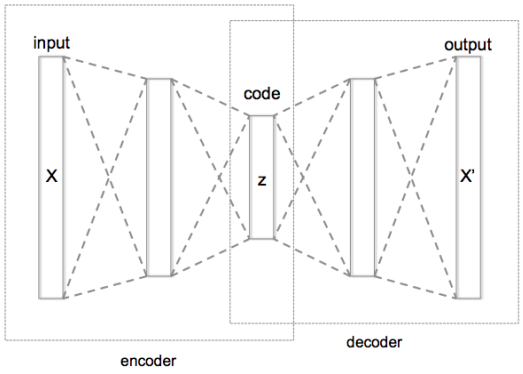
2.3.1 What is UNet?

In this part, I will try to elaborate what is UNet and how it works. Before UNet, I will begin with two other neural networks, autoencoder and ResNet, which have similar network structure to parts of UNet. In fact, these two helped me a lot to understand UNet, and hence I hope it will help you too. You can also skip these to go directly to UNet below if you know them already.

Autoencoder

As stated in Wiki, autoencoder is a network which learns to compress data from the input layer into a short code and then uncompress that code into something that closely matches the origin data. The schematic structure of an

autoencoder with 3 fully connected hidden layer is shown below.



In other words, the autoencoder is trying to learn an approximation to the identity function, so as to output $\hat{x}$ is similar to input x, such that

$$\hat{x} \approx x$$

From one point, the autoencoder often ends up learning a low-dimension representation very similar to PCAs when the number of hidden units in the middle of autoencoder is small. The autoencoder is useful for tasks such as object recognition and other vision tasks.

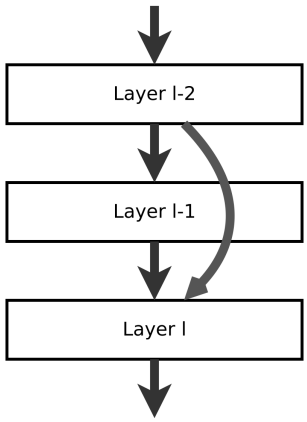
```
class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(256, 128),
            nn.Tanh(),
            nn.Linear(128, 64),
            nn.Tanh(),
            nn.Linear(64, 12),
            nn.Tanh(),
            nn.Linear(12, 3),   # compress to 3 features which can be visualized in␣
↪plt
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.Tanh(),
            nn.Linear(12, 64),
            nn.Tanh(),
            nn.Linear(64, 128),
            nn.Tanh(),
            nn.Linear(128, 256),
            nn.Sigmoid(),       # compress to a range (0, 1)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded
```

ResNet

The residual neural network (ResNet) is an artificial neural network (ANN) of a kind that builds on constructs known from pyramidal cells in the cerebral cortex. Residual neural networks do this by utilizing skip connections, or short-cuts to jump over some layers. And the reason for the skipping or short-cuts is to avoid the problem of vanishing gradients by reusing the values from previous layers. The structure of ResNet is shown below.



From my view, the ResNet also keep the previous learned features and in this way, it is good to work with tasks such as image denoising/impainting. Moreover, the simulations turn out that the short-cuts layers are functional to refine the image if they are added at the end of the network. For example, I have tried to add five convoluted res layer at the end and it did improve the quality of the image little.

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out
```

(continues on next page)

(continued from previous page)

```
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out


class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000):
        self.inplanes = 64
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(512 * block.expansion, num_classes)
```

(continues on next page)

(continued from previous page)

```
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

The above code is from Pytorch Vision https://pytorch.org/docs/0.4.0/_modules/torchvision/models/resnet.html .

UNet

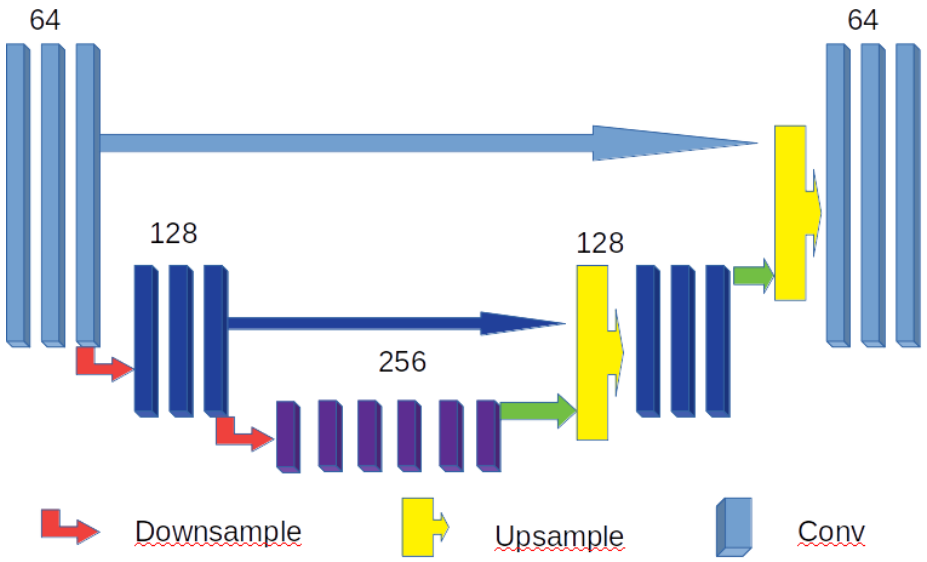
Finally, we will go into UNet. Facts: U-Net was created by Olaf Ronneberger, Philipp Fischer, Thomas Brox in 2015 at the paper "UNet: Convolutional Networks for Biomedical Image Segmentation". And until May 2019, it has been cited almost 6000, which shows little about how fast the development of deep learning.

The network(UNet) consists of a contracting path and an symmetric expansive path, which gives it the u-shaped architecture. The contracting path is a typical convolutional network that consists of repeated application of convolutions, each followed by a rectified linear unit (ReLU) and a max pooling operation. During the contraction, the spatial information is reduced while feature information is increased. The expansive pathway combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from

the contracting path.

From my view, the spatial information inherts the idea from the autoencoder while the difference is that it transfer the information into the feature spaces. While the concatenations of feature and spatial information is somehow inherts from ResNet while the difference is that it is not just short-cuts since it combines the spatial and feature information.

One example of UNet with two times- subsampling is elaborated below.



In our case, if we are doing image denoising/impainting. The UNet will encode the corrupted image into a lower dimensional space consisting of the essential features and decodes the essential features to the uncorrupted version, while UNet further combines spatial and feature information between the encoding and decoding sides of the network. These direct connections propagate feature representation to the decoder at each level and provide a shortcut for backpropagation.

2.3.2 How to write UNet in code?

The example code based on UNet in Pytorch is given below[for more details please refer to the code or the paper of UNet].

Structure of UNet

In the following code, the structure of UNet is given based on 3 downsamplings and 3 upsamplings, where 3 downsamplings are included in the Encoder network and 3 upsampling are included in the Decoder network. Five fully convoluted layers are added as the bottom layer of the network and two fully convoluted layers are added to the end of the network to further refine the output[improve the quality of the image].

Note: The number of up/down samplings can be selected as the parameters while tuning the network. It is worthy to try and explore.

And the dimentions of output also depend on the size of kernels/padding/dilation/input, and hence please be sure about the output based on different inputs.

In general, the dimensions of the output can be calculated as following, if we use `torch.nn.Conv2d`.

$$d_{out} = \frac{d_{in} + 2 * pad - dilation * (kernelsize - 1) - 1}{stride} + 1$$

where d_{out} is the dimension of output and d_{in} is the dimension of input.

And when it goes to the concatenation, we use `torch.nn.ConvTranspose2d`

$$d_{out} = (d_{in} - 1) * stide - 2 * pad + dilation * (kernelsize - 1) + outpad + 1$$

```
class UNet(torch.nn.Module):

    def __init__(self, opts):
        super(UNet, self).__init__()

        self.opts = opts
        input_channel_number = 5
        output_channel_number = 1
        kernel_size = 3

        # Encoder network
        self.down_block1 = UNet_down_block(input_channel_number, 64, False) # 64*520
        self.down_block2 = UNet_down_block(64, 128, True) # 64*520
        self.down_block3 = UNet_down_block(128, 256, True) # 64*260


        # bottom convolution
        self.mid_conv1 = torch.nn.Conv2d(256, 256, kernel_size, padding=(1, 1),␣
↪bias=False)# 64*260
        self.bn1 = Norm(256)
        self.mid_conv2 = torch.nn.Conv2d(256, 256, kernel_size, padding=(1, 1),␣
↪bias=False)# 64*260
        self.bn2 = Norm(256)
        self.mid_conv3 = torch.nn.Conv2d(256, 256, kernel_size, padding=(1, 1),␣
↪bias=False) #, dilation=4 # 64*260
        self.bn3 = Norm(256)
        self.mid_conv4 = torch.nn.Conv2d(256, 256, kernel_size, padding=(1, 1),␣
↪bias=False)# 64*260
        self.bn4 = Norm(256)
        self.mid_conv5 = torch.nn.Conv2d(256, 256, kernel_size, padding=(1, 1),␣
↪bias=False)# 64*260
        self.bn5 = Norm(256)

        # Decoder network
        self.up_block2 = UNet_up_block(128, 256, 128, 1)# 64*520
        self.up_block3 = UNet_up_block(64, 128, 64, 1)# 64*520

        # # Final output
        self.last_conv1 = torch.nn.Conv2d(64, 64, 3, padding=(1, 1), bias=False)#␣
↪64*520
        self.last_bn = Norm(64) #
        self.last_conv2 = torch.nn.Conv2d(64, output_channel_number, 3, padding=(1,␣
↪1))# 64*520
        self.last_bn2 = Norm(output_channel_number) # 64*520

        self.softplus = torch.nn.Softplus(beta=5)
        self.relu = torch.nn.ReLU()
```

(continues on next page)

(continued from previous page)

```
        self.tanhshrink = torch.nn.Tanhshrink()
        self.tanh = torch.nn.Tanh()


    def forward(self, x, test=False):
        x1 = self.down_block1(x)
        x2 = self.down_block2(x1)
        x3 = self.down_block3(x2)

        x4 = torch.nn.functional.leaky_relu(self.bn1(self.mid_conv1(x3)), 0.2)
        x4 = torch.nn.functional.leaky_relu(self.bn2(self.mid_conv2(x4)), 0.2)
        x4 = torch.nn.functional.leaky_relu(self.bn3(self.mid_conv3(x4)), 0.2)
        x4 = torch.nn.functional.leaky_relu(self.bn4(self.mid_conv4(x4)), 0.2)
        x4 = torch.nn.functional.leaky_relu(self.bn5(self.mid_conv5(x4)), 0.2)


        out = self.up_block2(x2, x4)
        out = self.up_block3(x1, out)

        out = torch.nn.functional.relu(self.last_conv1(out))
        out = self.last_conv2(out)
        out = self.softplus(out)
        return out
```

Details of Code

The upblock and downblock are defined as following:

```
class UNet_down_block(torch.nn.Module):

    def __init__(self, input_channel, output_channel, down_sample):
        super(UNet_down_block, self).__init__()
        kernel_size = 3
        self.conv1 = torch.nn.Conv2d(input_channel, output_channel, kernel_size,␣
↪stride=(1, 1), padding=(1, 1), bias=False)
        self.bn1 = Norm(output_channel)
        self.conv2 = torch.nn.Conv2d(output_channel, output_channel, kernel_size,␣
↪stride=(1, 1), padding=(1, 1), bias=False)
        self.bn2 = Norm(output_channel)
        self.conv3 = torch.nn.Conv2d(output_channel, output_channel, kernel_size,␣
↪stride=(1, 1), padding=(1, 1), bias=False)
        self.bn3 = Norm(output_channel)
        self.down_sampling = torch.nn.Conv2d(input_channel, input_channel, kernel_
↪size, stride=(2, 2), padding=(1, 1), bias=False)
        self.down_sample = down_sample


    def forward(self, x):
        if self.down_sample:
            x = self.down_sampling(x)
        # print('down',x.shape)
        x = torch.nn.functional.leaky_relu(self.bn1((self.conv1(x))), 0.2)
        x = torch.nn.functional.leaky_relu(self.bn2((self.conv2(x))), 0.2)
        x = torch.nn.functional.leaky_relu(self.bn3((self.conv3(x))), 0.2)
        # print(x.shape)
```

(continues on next page)

(continued from previous page)

```
        return x

class UNet_up_block(torch.nn.Module):

    def __init__(self, prev_channel, input_channel, output_channel, ID):
        super(UNet_up_block, self).__init__()
        kernel_size = 3
        self.ID = ID
        self.up_sampling = torch.nn.ConvTranspose2d(input_channel, input_channel, 4,␣
↪stride=(2, 2), padding=(1, 1))
        self.conv1 = torch.nn.Conv2d(prev_channel + input_channel, output_channel,␣
↪kernel_size, stride=(1, 1), padding=(1, 1), bias= False)
        self.bn1 = Norm(output_channel)
        self.conv2 = torch.nn.Conv2d(output_channel, output_channel, kernel_size,␣
↪stride=(1, 1), padding=(1, 1), bias= False)
        self.bn2 = Norm(output_channel)
        self.conv3 = torch.nn.Conv2d(output_channel, output_channel, kernel_size,␣
↪stride=(1, 1), padding=(1, 1), bias= False)
        self.bn3 = Norm(output_channel)


    def forward(self, prev_feature_map, x):

        if self.ID == 1:
            x = self.up_sampling(x)
        elif self.ID == 2:
            x = torch.nn.functional.interpolate(x, scale_factor=(2, 2), mode='nearest
↪')
        elif self.ID == 3:
            x = torch.nn.functional.interpolate(x, scale_factor=(2, 2), mode='area')
↪# 'nearest' | 'linear' | 'bilinear' | 'trilinear' | 'area'
        x = torch.cat((x, prev_feature_map), dim=1)
        # x = torch.nn.functional.leaky_relu(self.bn1((self.conv1(x))), 0.2)
        # x = torch.nn.functional.leaky_relu(self.bn2((self.conv2(x))), 0.2)
        # x = torch.nn.functional.leaky_relu(self.bn3((self.conv3(x))), 0.2)
        # print('up', x.shape)
        x = torch.nn.functional.leaky_relu((self.conv1(x)), 0.2)
        x = torch.nn.functional.leaky_relu((self.conv2(x)), 0.2)
        x = torch.nn.functional.leaky_relu((self.conv3(x)), 0.2)
        # print(x.shape)
        return x
```

Knowing the network, we can input the corrupted image, train the network, and output the cleaned image. Also note that the contracting path in UNet can be implemented based on differet kernels, say 3*3, 4*4 or 5*5.

Added May 29 2019: I also find a interesting summary of UNet here http://www.deeplearning.net/tutorial/unet.html .

Note: Plus, we just give the example of UNet while there are plenties of networks worthy to explore and try, such as VGG, ResNet, FrameletNet. Just google it, and you will find something worthy to try.

2.4 Train your network

In this part, we will focus on how to train our UNet. We will first cover several common loss functions and then go with how to train. Finally, we also give a method to further improve the training speed.

2.4.1 Loss functions

The loss function is the function to map an event or values of one or more to prepresent the cost associated with the event. And if we view the deep learning as an optimization problem, our goal will be to minimize the loss function.

To be simple, the loss function is a method to evaluate how well our neural network is and we use backward propogation to change the weights in the network to further improve our neural network or say to minimize our cost/loss.

Here, I will enumerate several common used loss functions based on Pytorch.

L1Loss

L1Loss creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

```
torch.nn.L1Loss()
```

$$l(x, y) = L = \{l_1, ..., l_N\}^T, l_n = |x_n - y_n|$$

where N is the batch size.

MSELoss

MSELoss creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

```
torch.nn.MSELoss()
```

$$l(x, y) = L = \{l_1, ..., l_N\}^T, l_n = (x_n - y_n)^2$$

MSSIM

MSSIM is the structural similarity(SSIM) index, which predicts the perceived quality of digital television and cinematic pictures, as well as other kinds of digital images and videos. To be simple, SSIM is used for measuring the similarity between two images, while it ranges in [0,1]. SSIM is designed to improve the traditional methods, while it works well while being combined with traditional methods, such as L1, L2, in deep learning.

```
MSSSIM(window_size=9, size_average=True)
```

Combinations

Try a combination of different loss functions.

```
Loss = L1Loss + MSSSIM
Loss = L1Loss + MSELoss + MSSSIM
```

Note: Here, only three loss functions are listed, and there are still a lot available there online. Google it! You can even define you own loss function based on your task. Network is flexible, and so with the loss functions. DIY!

2.4.2 Train, Train, Train

We start with the optimizer and then the structure of training.

Optimizer

The optimizer is the package implementing optimization algorithms. Most commonly used methods are already supported in Pytorch, and we can easily construct an optimizer object in pytorch by using one line of code. Two examples of optimizers are listed below.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

where the parameters of the optimizer can be set, such as the learning rate, weight decay and so on when initiating the optimizer.

From my view, the optimizer takes job of the backward propagation and optimizes the neural network based on certain optimization algorithm. The details of pytorch optim can be found https://pytorch.org/docs/stable/optim.html . To be simple, the optimizer change the weights in neural network to make it the optimal weights for mapping the input to target.

Backward Propagation

Training has been made really simple in Pytorch. As the example given in Pytorch, we only need to loop over the data iterator and feed the inputs into the neural network and optimize. The training is done!

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        badsino, goodsino = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = Unet(badsino)
        loss = criterion(outputs, goodsino)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
```

(continues on next page)

(continued from previous page)

```
print('Finished Training')
```

Note: We can train our neural network in GPU just simply transfering the tensors and the neural network onto the GPU.

```
network.cuda()
```

This would do all the work! That's amazing, right?

2.4.3 Warm Restart?(Optional)

The restart techniques are common in gradient-free optimizaion to deal with multimodal functions. This warm restarts borrows the idea from SGDR: Stochastic Gradient Descent with Warm Restarts.

The idea of the warm restart is to simulate a new warm-started run/restart of the optimizer once for every certain epochs.

Try to restart your optimizer:

```
if epoch % next_reset == 0:
    print("Resetting Optimizer\n")
    optimizer = torch.optim.Adam(network.network.parameters(), lr=self.opts.initial_
↪lr, betas=(0.5, 0.999))
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=self.
↪opts.lr_decay)
    network.set_optimizer(optimizer)

    # Set the next reset
    next_reset += self.opts.warm_reset_length + warm_reset_increment
    warm_reset_increment += self.opts.warm_reset_increment
```

Specially, from the paper, we can do the warm restarts that are not performed from scratch but emulated by decaying the learning rate.

$$lr = lr_{min} + \frac{1}{2}(lr_{max} - lr_{min})(1 + cos(\frac{T_{cur}}{T_i}\pi))$$

where lr_{min} and lr_{max} are the ranges for the learning rate, and T_{cur} accounts for how many epochs have been performed since the last restart, and T_i is the index of epochs.

Note: DIY warm starts.

2.5 Visualize your results!

2.5.1 Install Visdom and Open server

Visdom is a flexible tool for creating, organizing, and sharing visualizations of live, rich data. Supports Torch and Numpy, which can be found https://github.com/facebookresearch/visdom/.

From my view, visdom is similar to the TensorBoard from Tensorflow while it is still under developing I hope it will be much more strong in the future.

Install

Visdom can be easily installed by using pip.

```
pip install visdom
```

There are also other methods to install visdom which I am not familiar!

```
# Install Torch client
# (STABLE VERSION, NOT ALL CURRENT FEATURES ARE SUPPORTED)
luarocks install visdom
# Install visdom from source
pip install -e .
# If the above runs into issues, you can try the below
easy_install .

# Install Torch client from source (from th directory)
luarocks make
```

Open server

After install visdom, you can start server from command line by running

```
python -m visdom.server
```

Then, the visdom can be accessed by going to http://localhost:8097 in your browser, or your own host address if specified.

2.5.2 Project your output to Visdom

Now you have installed visdom, and next we will work on project our output to the server and then we can view them.

Step by step

First, we need to initate our visdom object by:

```
vis = visdom.Visdom()
```

Second, we need to open one window project our output:

```
window = None
```

Last, we need update our window with the output:

```
# project line
window = vis.line(X=np.arange(len(data)), Y=data, win=window, update='replace')
# project images
window = images(images, padding=5, win=window, nrow=2)
```

A summary of code

In the following code, I have generated three windows for loss of 'test', 'train', 'recon' and two windows for 'train_image', 'test_image':

```
import visdom
import numpy as np
# Start the server in terminal;
# #  visdom/ python -m visdom.server

class Visualize_Training():

    def __init__(self):
        self.vis = visdom.Visdom()
        self.win1 = None
        self.win3 = None
        self.win2 = None
        self.train_images = None
        self.test_images = None



    def Plot_Progress(self, path, window = "train"):
        '''
         Plot progress
        '''

        #TODO: Graph these on the same graph dummy!!!!!
        try:
            data = np.loadtxt(path)
            if window == "train":
                if self.win1 == None:
                    self.win1 = self.vis.line(X=np.arange(len(data)), Y=data,␣
↪opts=dict(xlabel='Historical Epoch',
                                       ylabel='Training Loss',
                                       title='Training Loss',
                                       legend=['Training Loss']))
                else:
                    self.vis.line(X=np.arange(len(data)), Y=data, win=self.win1,␣
↪update='replace')

            elif window == "test":
                if self.win3 == None:
                    self.win3 = self.vis.line(X=np.arange(len(data)), Y=data,␣
↪opts=dict(xlabel='Historical Epoch',
                                       ylabel='Testing Loss',
                                       title='Testing Loss',
                                       legend=['Testing Loss']))
                else:
                    self.vis.line(X=np.arange(len(data)), Y=data, win=self.win3,␣
↪update='replace')

            elif window == "recon":
                if self.win2 == None:
                    self.win2 = self.vis.line(X=np.arange(len(data)), Y=data,␣
↪opts=dict(xlabel='Historical Epoch',
                                       ylabel='Recon Loss',
                                       title='Recon Loss',
```

(continues on next page)

(continued from previous page)

```
                                    legend=['Recon Loss']))
                else:
                    self.vis.line(X=np.arange(len(data)), Y=data, win=self.win2,␣
↪update='replace')
        except:
            pass


    def Show_Train_Images(self, images, text='Images'):
        '''
        images: a list of same size images
        '''

        if self.train_images == None:

            self.train_images = self.vis.images(images, padding=5, nrow=2,␣
↪opts=dict(title=text))
        else:

            self.vis.images(images, padding=5, win=self.train_images, nrow=2,␣
↪opts=dict(title=text))

    def Show_Test_Images(self, images, text='Images'):
        '''
        images: a list of same size images
        '''

        if self.test_images == None:

            self.test_images = self.vis.images(images, padding=5, nrow=2,␣
↪opts=dict(title=text))
        else:

            self.vis.images(images, padding=5, win=self.test_images, nrow=2,␣
↪opts=dict(title=text))
```

Note: Try to generate your own visdom server!

2.6 Adjust your parameters!

In this part, I will try to explain about the problems I have confronted with and the parameters I have tuned when I try to train the neural network and also the methods I have found to view or fixed the problems. There must be a lot of other problems I can not cover here, and in this case google it first and see if other people have solved it.

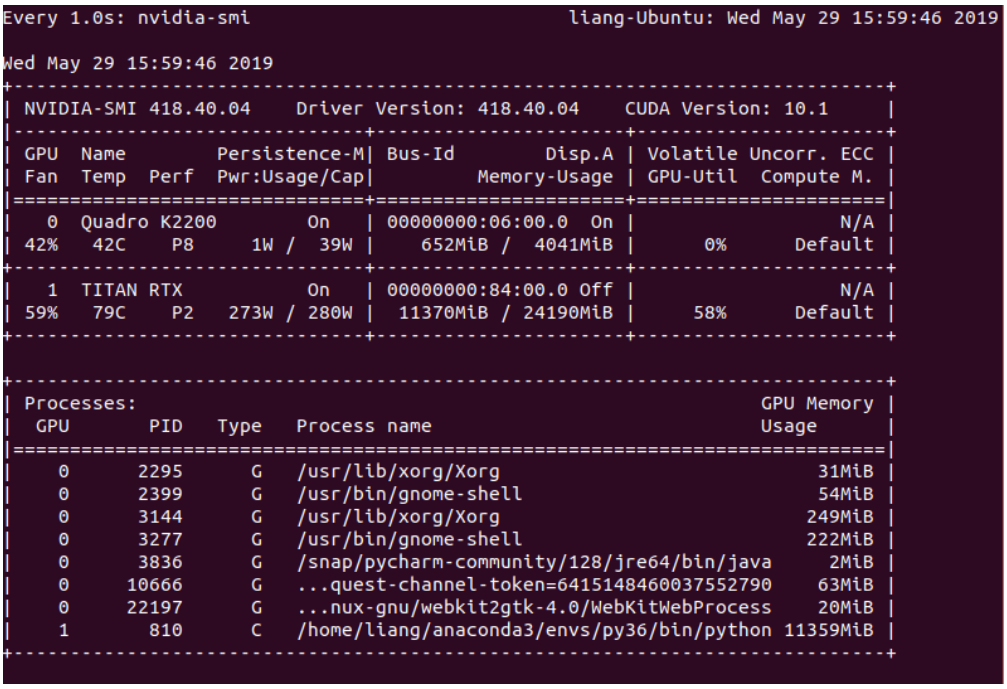
2.6.1 About memory and time consumption

Memory

Let's begin to view the usage[memory consumption] of GPU first, which can be easily done by calling the following code in terminal.

```
nvidia-smi
watch -n 1 nvidia-smi
```

Where nvidia-smi will show the results one time, by using *watch -n 1 nvidia-smi* we can let the terminal update the results every 1 second. One sample of the results are shown below.



Note: Keeping an eye on the memory consumption can help us do the memory management during the traing. For example, if our peak usage of memory is too high, we better reduce some parameters in training. e.g. the size of data input, the batchsize of data input, even the size of network[the width or depth of the neural network].

Time and memory

We can also check the memory consumption by using python wrapper. In the following, I have listed one example based on the *profile* function from memory profiler package, where both the time and memory consumption of program can be monitored by adding a wrapper in front of our target function.

```
from memory_profiler import profile

@profile()
def basic_mean(N=5):
   nbrs = list(range(0, 10 ** N))
   total = sum(nbrs)
   mean = sum(nbrs) / len(nbrs)
   return mean

if __name__ == '__main__':
    basic_mean()
```

After running

```
python -m memory_profiler profile.py
```

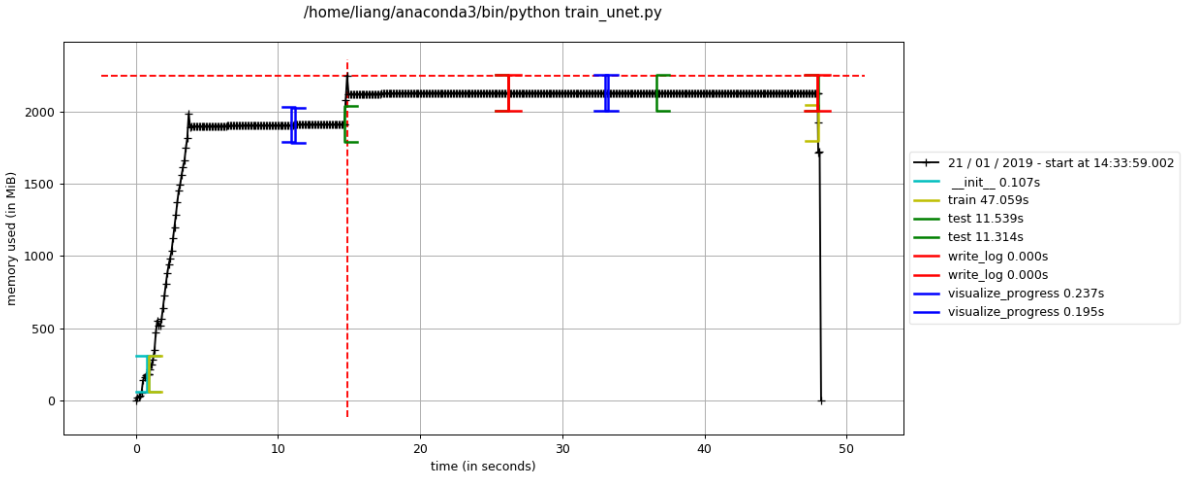
the log will be generated and shown in the terminal.

```
Line #    Mem usage    Increment   Line Contents
================================================
  2     36.7 MiB     36.7 MiB   @profile()
  3                             def basic_mean(N=5):
  4     40.5 MiB      3.7 MiB       nbrs = list(range(0, 10 ** N))
  5     40.5 MiB      0.0 MiB       total = sum(nbrs)
  6     40.5 MiB      0.0 MiB       mean = sum(nbrs) / len(nbrs)
  7     40.5 MiB      0.0 MiB       return mean
```

If we want to have the time-based memory usage, such as a report, we can run

```
mprof run <executable>
mprof plot
```

In this way, a recorded file with the time-based memory usage will be generated as following,



This is super cool, right?! But for real time checking, *watch -n 1 nvidia-smi* works better.

Note: There must be other methods, which I can not enumerate, please keep an eye on them too.

Next, let's go into more parameters dealing with not only the memory and time consumption but also the learning accuracy.

2.6.2 Parameters in data

If we are working on image denoising, the input are images with noise. In this case, the parameters we can change include but not limited to [the shape], [the distribution], [range/maximum/minimum/mean/median/deviation] of data. In this part, I would mainly focus on two of them, the shape and the distribution respectively.

Shape of data

When I am talking the shape of data, I mean the size of image. For example, each input of the neural network might be 3*256*256, which is a three-channel image with both width and height as 256. A bigger size of image will surely cost more operations and further affect the learning speed. Hence, choosing the right size or deciding the right resolution will be the first thing to track at the beginning or during the training.

E.g., we can scale the origin 3*256*256 to grayscale as 1*256*256, and we can further reduce the height and width by sampling origin image by 4:1 and obtain images with the size of 1*64*64.

Note: The options are flexible based on different targets.

Distribution of data

I have tried several most and found that mapping the range of origin images to [0,1] can achive my best performance. But the case might be different based on different tasks.

Normalization, as another method, is a technique often applied. The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. Normalization is required only when features have different ranges.

Note: For more details, please refer to https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029, which is a good article about normalization.

2.6.3 Parameters in optimizer

Deep learning neural networks are trained using the stochastic gradient descent, which is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation. Regarding to the otpimization, we will include two parameters which I have change frequently, inital learning rate and learning decay rate.

Inital Learning Rate

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. The learning rate may be the most important hyperparameter when configuring your neural network. Therefore it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior. For more details, please refer to https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

Note: Overall, a learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.

Learning Decay Rate

Learning Decay Rate is the amount that the learning rate are updated during training is referred to as the step size or the "learning rate." The learning decay rate is usually initated with a big value and then decay to a small value based on different algorithms. The reason for this is to make the weights of the network converge quickly to the solution at the begining and then converge more slow to reach the optimal solution when it is close to the solution.

2.6.4 Parameters in neural network

last but not the last, we can also tune parameters in neural network, UNet. Since most layers in UNet are equipped as the convolutional layer, then we can change the kernel size which can be managed to reduce the weight Convnets while making them deeper.

Kernel Size

The number of weights is dependent on the kernel size instead of the input size which is really important for images. Convolutional layers reduce memory usage and compute faster. For more, this article is a good reference -> https://blog.sicara.com/about-convolutional-layer-convolution-kernel-9a7325d34f7d

Other Parameters

There are also other things we could try, such as adding bias to fully connect layer, changing the number of upsampling and downsampling, trying different activation functions, revising the layers in up/down blocks and so on.

2.6.5 Try other neural networks

UNet is one of many neural networks for computer vision, and there are a lot of other networks online with open source models available in github. For example, Facebook research has post several modern models in https://github.com/pytorch/vision/tree/master/torchvision/models which include alexnet, densenet, googlenet, mobilenet, resnet, vgg. What we need to do is to change/revise certain parts of the networks and make it meet our requirements, and then the parameters tuning would be similar to what we have discussed above.

Note: Never stop in learning, because this area has been developed so fast.

Good Luck!

Liang, May 30, 2019

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search