misf-main/train.py

```
from main import main
main(mode=1)
```

misf-main/requirements.txt

```
numpy ~= 1.21
scipy ~= 1.0.1
future ~= 0.16.0
matplotlib ~= 2.2.2
pillow >= 6.2.0
opencv-python ~= 4.2.0.32
scikit-image ~= 0.14.0
pyaml
```

misf-main/test.py

```
from main import main
main(mode=2)
```

misf-main/main.py

```
import os
import cv2
import random
import numpy as np
import torch
import argparse
from shutil import copyfile
from src.config import Config
from src.misf import MISF
import torch.nn as nn


def main(mode=None):


    config = load_config(mode)


    # cuda visble devices
    os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(str(e) for e in config.GPU)


    # init device
    if torch.cuda.is_available():
        config.DEVICE = torch.device("cuda")
        torch.backends.cudnn.benchmark = True   # cudnn auto-tuner
    else:
        config.DEVICE = torch.device("cpu")



    # set cv2 running threads to 1 (prevents deadlocks with pytorch dataloader)
    cv2.setNumThreads(0)


    # initialize random seed
    torch.manual_seed(config.SEED)
    torch.cuda.manual_seed_all(config.SEED)
    np.random.seed(config.SEED)
    random.seed(config.SEED)



    # build the model and initialize
    model = MISF(config)
    model.load()

    iteration = model.inpaint_model.iteration
    if len(config.GPU) > 1:
        print('GPU:{}'.format(config.GPU))
        model.inpaint_model.generator = nn.DataParallel(model.inpaint_model.generator, config.GPU)
        model.inpaint_model.discriminator = nn.DataParallel(model.inpaint_model.discriminator, config.GP

    model.inpaint_model.iteration = iteration

    # print(model.inpaint_model)



    # model training
    if config.MODE == 1:
        # config.print()
        print('\nstart training...\n')
        model.train()

    # model test
    elif config.MODE == 2:
        print('\nstart testing...\n')
```

```
        model.test()


def load_config(mode=None):


    parser = argparse.ArgumentParser()
    parser.add_argument('--path', '--checkpoints', type=str, default='./checkpoints', help='model checkp

    args = parser.parse_args()
    config_path = os.path.join(args.path, 'config.yml')

    if not os.path.exists(args.path):
        os.makedirs(args.path)

    config = Config(config_path)

    # train mode
    if mode == 1:
        config.MODE = 1

    # test mode
    elif mode == 2:
        config.MODE = 2


    return config


if __name__ == "__main__":
    main()
```

misf-main/README.md

````
# MISF: Multi-level Interactive Siamese Filtering for High-Fidelity Image Inpainting in CVPR2022

We proposed a novel approach for high-fidelity image inpainting. Specifically, we use a single predictiv
<br><br>
![Framework](./images/frameworks.png)

## Prerequisites
- Python 3.7
- PyTorch >= 1.0 (test on PyTorch 1.0 and PyTorch 1.7.0)

## Dataset

- [Places2 Data of Places365-Standard](http://places2.csail.mit.edu/download.html)
- [CelebA](https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html)
- [Dunhuang]
- [Mask](https://drive.google.com/file/d/1cuw8QGfiop9b4K7yo5wPgPqXBIHjS6MI/view?usp=share_link)

1. For data folder path (CelebA) organize them as following:

```shell
--CelebA
   --train
      --1-1.png
   --valid
      --1-1.png
   --test
      --1-1.png
   --mask-train
■  --1-1.png
   --mask-valid
      --1-1.png
   --mask-test
      --0%-20%
        --1-1.png
      --20%-40%
        --1-1.png
      --40%-60%
        --1-1.png
 ```

 2. Run the code  `./data/data_list.py` to generate the data list

## Architecture details

<br><br>
![Framework](./images/misf_arch.png)

## Pretrained models

[CelebA](https://drive.google.com/drive/folders/14QVgtG5nbk5e00QRqEJBlBM5Q-aHF5Bd?usp=sharing)

[Places2](https://drive.google.com/drive/folders/14QVgtG5nbk5e00QRqEJBlBM5Q-aHF5Bd?usp=sharing)

[Dunhuang](https://drive.google.com/drive/folders/14QVgtG5nbk5e00QRqEJBlBM5Q-aHF5Bd?usp=sharing)

## Train

python train.py
<br>
For the parameters: checkpoints/config.yml

## Test

Such as test on the face dataset, please follow the following:
1. Make sure you have downloaded the "celebA_InpaintingModel_dis.pth" and "celebA_InpaintingModel_gen.pt
2. Change "MODEL_LOAD: celebA_InpaintingModel" in checkpoints/config.yml.
3. python test.py #For the parameters: checkpoints/config.yml
## Results
````

- Comparsion with SOTA, see paper for details.

More details are coming soon

Bibtex

```
@article{li2022misf,
  title={MISF: Multi-level Interactive Siamese Filtering for High-Fidelity Image Inpainting},
  author={Li, Xiaoguang and Guo, Qing and Lin, Di and Li, Ping and Feng, Wei and Wnag, Song},
  journal={CVPR},
  year={2022}
}
```

Acknowledgments
Parts of this code were derived from:
https://github.com/tsingqguo/efficientderain
https://github.com/knazeri/edge-connect

misf-main/src/config.py

```
import os
import yaml

class Config(dict):
    def __init__(self, config_path):
        with open(config_path, 'r') as f:
            self._yaml = f.read()
            self._dict = yaml.load(self._yaml)
            self._dict['PATH'] = os.path.dirname(config_path)

    def __getattr__(self, name):
        if self._dict.get(name) is not None:
            return self._dict[name]

        if DEFAULT_CONFIG.get(name) is not None:
            return DEFAULT_CONFIG[name]

        return None

    def print(self):
        print('Model configurations:')
        print('---------------------------------')
        print(self._yaml)
        print('')
        print('---------------------------------')
        print('')


DEFAULT_CONFIG = {
    'NMS': 1,
    'SEED': 10,                     # random seed
    'GPU': [0],                     # list of gpu ids
    'DEBUG': 0,                     # turns on debugging mode
    'VERBOSE': 0,                   # turns on verbose mode in the output console

    'LR': 0.0001,                   # learning rate
    'D2G_LR': 0.1,                  # discriminator/generator learning rate ratio
    'BETA1': 0.0,                   # adam optimizer beta1
    'BETA2': 0.9,                   # adam optimizer beta2
    'BATCH_SIZE': 8,                # input batch size for training
    'INPUT_SIZE': 256,              # input image size for training 0 for original size
    'MAX_ITERS': 2e6,               # maximum number of iterations to train the model

    'L1_LOSS_WEIGHT': 1,            # l1 loss weight
    'FM_LOSS_WEIGHT': 10,           # feature-matching loss weight
    'STYLE_LOSS_WEIGHT': 1,         # style loss weight
    'CONTENT_LOSS_WEIGHT': 1,       # perceptual loss weight
    'INPAINT_ADV_LOSS_WEIGHT': 0.01,# adversarial loss weight

    'GAN_LOSS': 'nsgan',            # nsgan | lsgan | hinge
    'GAN_POOL_SIZE': 0,             # fake images pool size

    'SAVE_INTERVAL': 1000,          # how many iterations to wait before saving model (0: never)
    'SAMPLE_INTERVAL': 1000,        # how many iterations to wait before sampling (0: never)
    'SAMPLE_SIZE': 12,              # number of images to sample
    'EVAL_INTERVAL': 0,             # how many iterations to wait before model evaluation (0: never)
    'LOG_INTERVAL': 10,             # how many iterations to wait before logging training status (0: nev
}
```

misf-main/src/utils.py

```
import os
import sys
import time
import random
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


def create_dir(dir):
    if not os.path.exists(dir):
        os.makedirs(dir)


def create_mask(width, height, mask_width, mask_height, x=None, y=None):
    mask = np.zeros((height, width))
    mask_x = x if x is not None else random.randint(0, width - mask_width)
    mask_y = y if y is not None else random.randint(0, height - mask_height)
    mask[mask_y:mask_y + mask_height, mask_x:mask_x + mask_width] = 1
    return mask


def stitch_images(inputs, *outputs, img_per_row=2):
    gap = 5
    columns = len(outputs) + 1

    width, height = inputs[0][:, :, 0].shape
    img = Image.new('RGB', (width * img_per_row * columns + gap * (img_per_row - 1), height * int(len(in
    images = [inputs, *outputs]

    for ix in range(len(inputs)):
        xoffset = int(ix % img_per_row) * width * columns + int(ix % img_per_row) * gap
        yoffset = int(ix / img_per_row) * height

        for cat in range(len(images)):
            im = np.array((images[cat][ix]).cpu()).astype(np.uint8).squeeze()
            im = Image.fromarray(im)
            img.paste(im, (xoffset + cat * width, yoffset))

    return img


def imshow(img, title=''):
    fig = plt.gcf()
    fig.canvas.set_window_title(title)
    plt.axis('off')
    plt.imshow(img, interpolation='none')
    plt.show()


def imsave(img, path):
    im = Image.fromarray(img.cpu().numpy().astype(np.uint8).squeeze())
    im.save(path)


class Progbar(object):

    def __init__(self, target, width=25, verbose=1, interval=0.05,
                 stateful_metrics=None):
        self.target = target
        self.width = width
        self.verbose = verbose
        self.interval = interval
        if stateful_metrics:
            self.stateful_metrics = set(stateful_metrics)
        else:
            self.stateful_metrics = set()
        self._dynamic_display = ((hasattr(sys.stdout, 'isatty') and
```

```
                             sys.stdout.isatty()) or
                            'ipykernel' in sys.modules or
                            'posix' in sys.modules)
        self._total_width = 0
        self._seen_so_far = 0
        # We use a dict + list to avoid garbage collection
        # issues found in OrderedDict
        self._values = {}
        self._values_order = []
        self._start = time.time()
        self._last_update = 0

    def update(self, current, values=None):

        values = values or []
        for k, v in values:
            if k not in self._values_order:
                self._values_order.append(k)
            if k not in self.stateful_metrics:
                if k not in self._values:
                    self._values[k] = [v * (current - self._seen_so_far),
                                       current - self._seen_so_far]
                else:
                    self._values[k][0] += v * (current - self._seen_so_far)
                    self._values[k][1] += (current - self._seen_so_far)
            else:
                self._values[k] = v
        self._seen_so_far = current

        now = time.time()
        info = ' - %.0fs' % (now - self._start)
        if self.verbose == 1:
            if (now - self._last_update < self.interval and
                    self.target is not None and current < self.target):
                return

            prev_total_width = self._total_width
            if self._dynamic_display:
                sys.stdout.write('\b' * prev_total_width)
                sys.stdout.write('\r')
            else:
                sys.stdout.write('\n')

            if self.target is not None:
                numdigits = int(np.floor(np.log10(self.target))) + 1
                barstr = '%%%dd/%d [' % (numdigits, self.target)
                bar = barstr % current
                prog = float(current) / self.target
                prog_width = int(self.width * prog)
                if prog_width > 0:
                    bar += ('=' * (prog_width - 1))
                    if current < self.target:
                        bar += '>'
                    else:
                        bar += '='
                bar += ('.' * (self.width - prog_width))
                bar += ']'
            else:
                bar = '%7d/Unknown' % current

            self._total_width = len(bar)
            sys.stdout.write(bar)

            if current:
                time_per_unit = (now - self._start) / current
            else:
                time_per_unit = 0
            if self.target is not None and current < self.target:
                eta = time_per_unit * (self.target - current)
                if eta > 3600:
                    eta_format = '%d:%02d:%02d' % (eta // 3600,
                                                   (eta % 3600) // 60,
```

```
                                                eta % 60)
                elif eta > 60:
                    eta_format = '%d:%02d' % (eta // 60, eta % 60)
                else:
                    eta_format = '%ds' % eta

                info = ' - ETA: %s' % eta_format
            else:
                if time_per_unit >= 1:
                    info += ' %.0fs/step' % time_per_unit
                elif time_per_unit >= 1e-3:
                    info += ' %.0fms/step' % (time_per_unit * 1e3)
                else:
                    info += ' %.0fus/step' % (time_per_unit * 1e6)

            for k in self._values_order:
                info += ' - %s:' % k
                if isinstance(self._values[k], list):
                    avg = np.mean(self._values[k][0] / max(1, self._values[k][1]))
                    if abs(avg) > 1e-3:
                        info += ' %.4f' % avg
                    else:
                        info += ' %.4e' % avg
                else:
                    info += ' %s' % self._values[k]

            self._total_width += len(info)
            if prev_total_width > self._total_width:
                info += (' ' * (prev_total_width - self._total_width))

            if self.target is not None and current >= self.target:
                info += '\n'

            sys.stdout.write(info)
            sys.stdout.flush()

        elif self.verbose == 2:
            if self.target is None or current >= self.target:
                for k in self._values_order:
                    info += ' - %s:' % k
                    avg = np.mean(self._values[k][0] / max(1, self._values[k][1]))
                    if avg > 1e-3:
                        info += ' %.4f' % avg
                    else:
                        info += ' %.4e' % avg
                info += '\n'

                sys.stdout.write(info)
                sys.stdout.flush()

        self._last_update = now

    def add(self, n, values=None):
        self.update(self._seen_so_far + n, values)
```

misf-main/kpn/network.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# ----------------------------------------
#         Initialize the networks
# ----------------------------------------
def weights_init(net, init_type = 'normal', init_gain = 0.02):

    def init_func(m):
        classname = m.__class__.__name__
        if hasattr(m, 'weight') and classname.find('Conv') != -1:
            if init_type == 'normal':
                torch.nn.init.normal_(m.weight.data, 0.0, init_gain)
            elif init_type == 'xavier':
                torch.nn.init.xavier_normal_(m.weight.data, gain = init_gain)
            elif init_type == 'kaiming':
                torch.nn.init.kaiming_normal_(m.weight.data, a = 0, mode = 'fan_in')
            elif init_type == 'orthogonal':
                torch.nn.init.orthogonal_(m.weight.data, gain = init_gain)
            else:
                raise NotImplementedError('initialization method [%s] is not implemented' % init_type)
        elif classname.find('BatchNorm2d') != -1:
            torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
            torch.nn.init.constant_(m.bias.data, 0.0)

    # apply the initialization function <init_func>
    print('initialize network with %s type' % init_type)
    net.apply(init_func)

# ----------------------------------------
#      Kernel Prediction Network (KPN)
# ----------------------------------------
class Basic(nn.Module):
    def __init__(self, in_ch, out_ch, g=16, channel_att=False, spatial_att=False):
        super(Basic, self).__init__()
        self.channel_att = channel_att
        self.spatial_att = spatial_att
        self.conv1 = nn.Sequential(
                nn.Conv2d(in_channels=in_ch, out_channels=out_ch, kernel_size=3, stride=1, padding=1),
                # nn.BatchNorm2d(out_ch),
                nn.ReLU(),
                nn.Conv2d(in_channels=out_ch, out_channels=out_ch, kernel_size=3, stride=1, padding=1),
                # nn.BatchNorm2d(out_ch),
                nn.ReLU(),
                nn.Conv2d(in_channels=out_ch, out_channels=out_ch, kernel_size=3, stride=1, padding=1),
                # nn.BatchNorm2d(out_ch),
                nn.ReLU()
            )

        if channel_att:
            self.att_c = nn.Sequential(
                nn.Conv2d(2*out_ch, out_ch//g, 1, 1, 0),
                nn.ReLU(),
                nn.Conv2d(out_ch//g, out_ch, 1, 1, 0),
                nn.Sigmoid()
            )
        if spatial_att:
            self.att_s = nn.Sequential(
                nn.Conv2d(in_channels=2, out_channels=1, kernel_size=7, stride=1, padding=3),
                nn.Sigmoid()
            )

    def forward(self, data):
        """
        Forward function.
        :param data:
```

```
        :return: tensor
        """
        fm = self.conv1(data)
        if self.channel_att:
            # fm_pool = F.adaptive_avg_pool2d(fm, (1, 1)) + F.adaptive_max_pool2d(fm, (1, 1))
            fm_pool = torch.cat([F.adaptive_avg_pool2d(fm, (1, 1)), F.adaptive_max_pool2d(fm, (1, 1))],
            att = self.att_c(fm_pool)
            fm = fm * att
        if self.spatial_att:
            fm_pool = torch.cat([torch.mean(fm, dim=1, keepdim=True), torch.max(fm, dim=1, keepdim=True)
            att = self.att_s(fm_pool)
            fm = fm * att
        return fm

class KPN(nn.Module):
    def __init__(self, kernel_size=[3], sep_conv=False, channel_att=False, spatial_att=False, upMode='bi
        super(KPN, self).__init__()
        self.upMode = upMode
        self.core_bias = core_bias
        self.kernel_size = kernel_size

        in_channel = 4
        out_channel = 64 * (self.kernel_size[0] ** 2)

        self.conv1 = Basic(in_channel, 64, channel_att=False, spatial_att=False) # 256*256
        self.conv2 = Basic(64, 128, channel_att=False, spatial_att=False)        # 128*128
        self.conv3 = Basic(128 + 128, 256, channel_att=False, spatial_att=False)  # 64*64

        self.conv4 = Basic(256, 512, channel_att=False, spatial_att=False)

        self.conv7 = Basic(256 + 512, 256, channel_att=channel_att, spatial_att=spatial_att)
        self.conv8 = Basic(256 + 256, 128, channel_att=channel_att, spatial_att=spatial_att)
        self.conv9 = Basic(128 + 64, 64, channel_att=channel_att, spatial_att=spatial_att)

        self.kernels = nn.Conv2d(256, out_channel, 1, 1, 0)

        out_channel_img = 3 * (self.kernel_size[0] ** 2)
        self.core_img = nn.Conv2d(64, out_channel_img, 1, 1, 0)

        self.kernel_pred = KernelConv(kernel_size, sep_conv, self.core_bias)

        self.conv_final = nn.Conv2d(in_channels=12, out_channels=3, kernel_size=3, stride=1, padding=1)

        self.iteration = 0

    def forward(self, data_with_est, x):

        conv1 = self.conv1(data_with_est) #64*256*256
        conv2 = self.conv2(F.avg_pool2d(conv1, kernel_size=2, stride=2)) # 128*128*128

        conv2 = torch.cat([conv2, x], dim=1)

        conv3 = self.conv3(F.avg_pool2d(conv2, kernel_size=2, stride=2)) # 256*64*64
        kernels = self.kernels(conv3)
        kernels = kernels.unsqueeze(dim=0)

#       kernels = F.interpolate(input=kernels, size=(256*9, 64, 64), mode='nearest')
        kernels = F.interpolate(input=kernels, size=(256*9, data_with_est.shape[-1]//4, data_with_est.sh

        kernels = kernels.squeeze(dim=0)


        conv4 = self.conv4(conv3)

        conv7 = self.conv7(torch.cat([conv3, conv4], dim=1))
        conv8 = self.conv8(torch.cat([conv2, F.interpolate(conv7, scale_factor=2, mode=self.upMode)], di
        conv9 = self.conv9(torch.cat([conv1, F.interpolate(conv8, scale_factor=2, mode=self.upMode)], di
        core_img = self.core_img(conv9)

        return kernels, core_img
class KernelConv(nn.Module):
    """
```

```
the class of computing prediction
"""
def __init__(self, kernel_size=[5], sep_conv=False, core_bias=False):
    super(KernelConv, self).__init__()
    self.kernel_size = sorted(kernel_size)
    self.sep_conv = sep_conv
    self.core_bias = core_bias

def _sep_conv_core(self, core, batch_size, N, color, height, width):
    """
    convert the sep_conv core to conv2d core
    2p --> p^2
    :param core: shape: batch*(N*2*K)*height*width
    :return:
    """
    kernel_total = sum(self.kernel_size)
    core = core.view(batch_size, N, -1, color, height, width)
    if not self.core_bias:
        core_1, core_2 = torch.split(core, kernel_total, dim=2)
    else:
        core_1, core_2, core_3 = torch.split(core, kernel_total, dim=2)
    # output core
    core_out = {}
    cur = 0
    for K in self.kernel_size:
        t1 = core_1[:, :, cur:cur + K, ...].view(batch_size, N, K, 1, 3, height, width)
        t2 = core_2[:, :, cur:cur + K, ...].view(batch_size, N, 1, K, 3, height, width)
        core_out[K] = torch.einsum('ijklno,ijlmno->ijkmno', [t1, t2]).view(batch_size, N, K * K, col
        cur += K
    # it is a dict
    return core_out, None if not self.core_bias else core_3.squeeze()

def _convert_dict(self, core, batch_size, N, color, height, width):
    """
    make sure the core to be a dict, generally, only one kind of kernel size is suitable for the fun
    :param core: shape: batch_size*(N*K*K)*height*width
    :return: core_out, a dict
    """
    core_out = {}
    core = core.view(batch_size, N, -1, color, height, width)
    core_out[self.kernel_size[0]] = core[:, :, 0:self.kernel_size[0]**2, ...]
    bias = None if not self.core_bias else core[:, :, -1, ...]
    return core_out, bias

def forward(self, frames, core, white_level=1.0, rate=1):
    """
    compute the pred image according to core and frames
    :param frames: [batch_size, N, 3, height, width]
    :param core: [batch_size, N, dict(kernel), 3, height, width]
    :return:
    """
    if len(frames.size()) == 5:
        batch_size, N, color, height, width = frames.size()
    else:
        batch_size, N, height, width = frames.size()
        color = 1
        frames = frames.view(batch_size, N, color, height, width)
    if self.sep_conv:
        core, bias = self._sep_conv_core(core, batch_size, N, color, height, width)
    else:
        core, bias = self._convert_dict(core, batch_size, N, color, height, width)
    img_stack = []
    pred_img = []
    kernel = self.kernel_size[::-1]
    for index, K in enumerate(kernel):
        if not img_stack:
            padding_num = (K//2) * rate
            frame_pad = F.pad(frames, [padding_num, padding_num, padding_num, padding_num])
            for i in range(0, K):
                for j in range(0, K):
                    img_stack.append(frame_pad[..., i*rate:i*rate + height, j*rate:j*rate + width])
            img_stack = torch.stack(img_stack, dim=2)
```

```
            else:
                k_diff = (kernel[index - 1] - kernel[index]) // 2
                img_stack = img_stack[:, :, k_diff:-k_diff, ...]
            # print('img_stack:', img_stack.size())
            pred_img.append(torch.sum(
                core[K].mul(img_stack), dim=2, keepdim=False
            ))
        pred_img = torch.stack(pred_img, dim=0)
        # print('pred_stack:', pred_img.size())
        pred_img_i = torch.mean(pred_img, dim=0, keepdim=False)
        #print("pred_img_i", pred_img_i.size())
        # N = 1
        pred_img_i = pred_img_i.squeeze(2)
        #print("pred_img_i", pred_img_i.size())
        # if bias is permitted
        if self.core_bias:
            if bias is None:
                raise ValueError('The bias should not be None.')
            pred_img_i += bias
        # print('white_level', white_level.size())
        pred_img_i = pred_img_i / white_level
        #pred_img = torch.mean(pred_img_i, dim=1, keepdim=True)
        # print('pred_img:', pred_img.size())
        # print('pred_img_i:', pred_img_i.size())
        return pred_img_i

class LossFunc(nn.Module):
    """
    loss function of KPN
    """
    def __init__(self, coeff_basic=1.0, coeff_anneal=1.0, gradient_L1=True, alpha=0.9998, beta=100):
        super(LossFunc, self).__init__()
        self.coeff_basic = coeff_basic
        self.coeff_anneal = coeff_anneal
        self.loss_basic = LossBasic(gradient_L1)
        self.loss_anneal = LossAnneal(alpha, beta)

    def forward(self, pred_img_i, pred_img, ground_truth, global_step):
        """
        forward function of loss_func
        :param frames: frame_1 ~ frame_N, shape: [batch, N, 3, height, width]
        :param core: a dict coverted by ......
        :param ground_truth: shape [batch, 3, height, width]
        :param global_step: int
        :return: loss
        """
        return self.coeff_basic * self.loss_basic(pred_img, ground_truth), self.coeff_anneal * self.loss

class LossBasic(nn.Module):
    """
    Basic loss function.
    """
    def __init__(self, gradient_L1=True):
        super(LossBasic, self).__init__()
        self.l1_loss = nn.L1Loss()
        self.l2_loss = nn.MSELoss()
        self.gradient = TensorGradient(gradient_L1)

    def forward(self, pred, ground_truth):
        return self.l2_loss(pred, ground_truth) + \
               self.l1_loss(self.gradient(pred), self.gradient(ground_truth))

class LossAnneal(nn.Module):
    """
    anneal loss function
    """
    def __init__(self, alpha=0.9998, beta=100):
        super(LossAnneal, self).__init__()
        self.global_step = 0
        self.loss_func = LossBasic(gradient_L1=True)
        self.alpha = alpha
        self.beta = beta
```

```
    def forward(self, global_step, pred_i, ground_truth):
        """
        :param global_step: int
        :param pred_i: [batch_size, N, 3, height, width]
        :param ground_truth: [batch_size, 3, height, width]
        :return:
        """
        loss = 0
        for i in range(pred_i.size(1)):
            loss += self.loss_func(pred_i[:, i, ...], ground_truth)
        loss /= pred_i.size(1)
        return self.beta * self.alpha ** global_step * loss

class TensorGradient(nn.Module):
    """
    the gradient of tensor
    """
    def __init__(self, L1=True):
        super(TensorGradient, self).__init__()
        self.L1 = L1

    def forward(self, img):
        w, h = img.size(-2), img.size(-1)
        l = F.pad(img, [1, 0, 0, 0])
        r = F.pad(img, [0, 1, 0, 0])
        u = F.pad(img, [0, 0, 1, 0])
        d = F.pad(img, [0, 0, 0, 1])
        if self.L1:
            return torch.abs((l - r)[..., 0:w, 0:h]) + torch.abs((u - d)[..., 0:w, 0:h])
        else:
            return torch.sqrt(
                torch.pow((l - r)[..., 0:w, 0:h], 2) + torch.pow((u - d)[..., 0:w, 0:h], 2)
            )

if __name__ == '__main__':

    kpn = KPN().cuda()
    a = torch.randn(4, 3, 224, 224).cuda()
    b = kpn(a, a)
    print(b.shape)
```

misf-main/src/networks.py

```
import torch
import torch.nn as nn
from kpn.network import KernelConv
import kpn.utils as kpn_utils
import numpy as np


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

    def init_weights(self, init_type='normal', gain=0.02):
        def init_func(m):
            classname = m.__class__.__name__
            if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear') != -1):
                if init_type == 'normal':
                    nn.init.normal_(m.weight.data, 0.0, gain)
                elif init_type == 'xavier':
                    nn.init.xavier_normal_(m.weight.data, gain=gain)
                elif init_type == 'kaiming':
                    nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
                elif init_type == 'orthogonal':
                    nn.init.orthogonal_(m.weight.data, gain=gain)

                if hasattr(m, 'bias') and m.bias is not None:
                    nn.init.constant_(m.bias.data, 0.0)

            elif classname.find('BatchNorm2d') != -1:
                nn.init.normal_(m.weight.data, 1.0, gain)
                nn.init.constant_(m.bias.data, 0.0)

        self.apply(init_func)


class InpaintGenerator(BaseNetwork):
    def __init__(self, config=None, residual_blocks=8, init_weights=True):
        super(InpaintGenerator, self).__init__()

        self.filter_type = config.FILTER_TYPE
        self.kernel_size = config.kernel_size

        self.encoder0 = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(in_channels=4, out_channels=64, kernel_size=7, padding=0),
            nn.InstanceNorm2d(64, track_running_stats=False),
            nn.ReLU(True)
        )

        self.encoder1 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128, track_running_stats=False),
            nn.ReLU(True)
        )

        self.encoder2 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256, track_running_stats=False),
            nn.ReLU(True)
        )

        blocks = []
        for _ in range(residual_blocks):
            block = ResnetBlock(256, 2)
            blocks.append(block)

        self.middle = nn.Sequential(*blocks)

        self.decoder = nn.Sequential(
```

```
            nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128, track_running_stats=False),
            nn.ReLU(True),

            nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(64, track_running_stats=False),
            nn.ReLU(True),

            nn.ReflectionPad2d(3),
            nn.Conv2d(in_channels=64, out_channels=3, kernel_size=7, padding=0),
        )

        self.kernel_pred = KernelConv(kernel_size=[3], sep_conv=False, core_bias=False)

        self.kpn_model = kpn_utils.create_generator()

        if init_weights:
            self.init_weights()

    def forward(self, x):
        inputs = x.clone()

        x = self.encoder0(x) # 64*256*256
        x = self.encoder1(x) # 128*128*128

        kernels, kernels_img = self.kpn_model(inputs, x)

        x = self.encoder2(x) # 256*64*64
        x = self.kernel_pred(x, kernels, white_level=1.0, rate=1)

        x = self.middle(x) # 256*64*64

        x = self.decoder(x) # 3*256*256

        x = self.kernel_pred(x, kernels_img, white_level=1.0, rate=1)

        x = (torch.tanh(x) + 1) / 2

        return x

    def save_feature(self, x, name):
        x = x.cpu().numpy()
        np.save('./result/{}'.format(name), x)


class Discriminator(BaseNetwork):
    def __init__(self, in_channels, use_sigmoid=True, use_spectral_norm=True, init_weights=True):
        super(Discriminator, self).__init__()
        self.use_sigmoid = use_sigmoid

        self.conv1 = self.features = nn.Sequential(
            spectral_norm(nn.Conv2d(in_channels=in_channels, out_channels=64, kernel_size=4, stride=2, p
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.conv2 = nn.Sequential(
            spectral_norm(nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.conv3 = nn.Sequential(
            spectral_norm(nn.Conv2d(in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.conv4 = nn.Sequential(
            spectral_norm(nn.Conv2d(in_channels=256, out_channels=512, kernel_size=4, stride=1, padding=
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.conv5 = nn.Sequential(
            spectral_norm(nn.Conv2d(in_channels=512, out_channels=1, kernel_size=4, stride=1, padding=1,
```

```
        )

        if init_weights:
            self.init_weights()

    def forward(self, x):
        conv1 = self.conv1(x)
        conv2 = self.conv2(conv1)
        conv3 = self.conv3(conv2)
        conv4 = self.conv4(conv3)
        conv5 = self.conv5(conv4)

        outputs = conv5
        if self.use_sigmoid:
            outputs = torch.sigmoid(conv5)

        return outputs, [conv1, conv2, conv3, conv4, conv5]


class ResnetBlock(nn.Module):
    def __init__(self, dim, dilation=1, use_spectral_norm=False):
        super(ResnetBlock, self).__init__()
        self.conv_block = nn.Sequential(
            nn.ReflectionPad2d(dilation),
            spectral_norm(nn.Conv2d(in_channels=dim, out_channels=dim, kernel_size=3, padding=0, dilatio
            nn.InstanceNorm2d(dim, track_running_stats=False),
            nn.ReLU(True),

            nn.ReflectionPad2d(1),
            spectral_norm(nn.Conv2d(in_channels=dim, out_channels=dim, kernel_size=3, padding=0, dilatio
            nn.InstanceNorm2d(dim, track_running_stats=False),
        )

    def forward(self, x):
        out = x + self.conv_block(x)

        return out


def spectral_norm(module, mode=True):
    if mode:
        return nn.utils.spectral_norm(module)

    return module
```

misf-main/src/dataset.py

```
import json
import os
import random

import numpy as np
import scipy
import torch
import torchvision.transforms.functional as F
from PIL import Image
from scipy.misc import imread
from skimage.color import rgb2gray, gray2rgb

from torch.utils.data import DataLoader



class Dataset(torch.utils.data.Dataset):
    def __init__(self, config, flist, mask_flist, augment=True, training=True):
        super(Dataset, self).__init__()
        self.augment = augment
        self.training = training
        self.data = self.load_flist(flist)
        self.mask_data = self.load_flist(mask_flist)

        self.input_size = config.INPUT_SIZE
        self.sigma = config.SIGMA
        self.mask = config.MASK
        self.nms = config.NMSMASK_REVERSE

        self.reverse_mask = config.MASK_REVERSE
        self.mask_threshold = config.MASK_THRESHOLD

        print('training:{}  mask:{}  mask_list:{}  data_list:{}'.format(training, self.mask, mask_flist,

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        try:
            item = self.load_item(index)
        except:
            print('loading error: ' + self.data[index])
            item = self.load_item(0)

        return item

    def load_name(self, index):
        name = self.data[index]
        return os.path.basename(name)

    def load_item(self, index):

        size = self.input_size

        # load image
        img = imread(self.data[index])

        # gray to rgb
        if len(img.shape) < 3:
            img = gray2rgb(img)

        # resize/crop if needed
        if size != 0:
            img = self.resize(img, size, size)

        # load mask
        mask = self.load_mask(img, index % len(self.mask_data))
        if self.reverse_mask == 1:
```

```
        mask = 255 - mask


    # augment data
    if self.augment and np.random.binomial(1, 0.5) > 0:
        img = img[:, ::-1, ...]
        mask = mask[:, ::-1, ...]


    return self.to_tensor(img), self.to_tensor(mask)


def load_mask(self, img, index):
    imgh, imgw = img.shape[0:2]
    mask_type = self.mask

    if self.training:
        mask_index = random.randint(0, len(self.mask_data) - 1)
    else:
        mask_index = index
        print('+++++++++++++++')

    mask = imread(self.mask_data[mask_index])
    mask = self.resize(mask, imgh, imgw)
    mask = (mask > self.mask_threshold).astype(np.uint8) * 255       # threshold due to interpolatio


    return mask

def to_tensor(self, img):
    img = Image.fromarray(img)
    img_t = F.to_tensor(img).float()
    return img_t

def resize(self, img, height, width, centerCrop=True):
    imgh, imgw = img.shape[0:2]

    if centerCrop and imgh != imgw:
        # center crop
        side = np.minimum(imgh, imgw)
        j = (imgh - side) // 2
        i = (imgw - side) // 2
        img = img[j:j + side, i:i + side, ...]

    img = scipy.misc.imresize(img, [height, width])

    return img

def load_flist(self, flist):
    if flist is None:
        return []
    with open(flist, 'r') as j:
        f_list = json.load(j)
        return f_list


def create_iterator(self, batch_size):
    while True:
        sample_loader = DataLoader(
            dataset=self,
            batch_size=batch_size,
            drop_last=True
        )

        for item in sample_loader:
            yield item
```

misf-main/src/metrics.py

```
import torch
import torch.nn as nn


class PSNR(nn.Module):
    def __init__(self, max_val):
        super(PSNR, self).__init__()

        base10 = torch.log(torch.tensor(10.0))
        max_val = torch.tensor(max_val).float()

        self.register_buffer('base10', base10)
        self.register_buffer('max_val', 20 * torch.log(max_val) / base10)

    def __call__(self, a, b):
        mse = torch.mean((a.float() - b.float()) ** 2)

        if mse == 0:
            return torch.tensor(0)

        return self.max_val - 10 * torch.log(mse) / self.base10
```

misf-main/src/__init__.py

```
# empty
```

misf-main/src/models.py

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from .networks import InpaintGenerator, Discriminator
from .loss import AdversarialLoss, PerceptualLoss, StyleLoss


class BaseModel(nn.Module):
    def __init__(self, name, config):
        super(BaseModel, self).__init__()

        self.name = name
        self.config = config
        self.iteration = 0

        self.model_save = config.PATH


    def load(self, type):
        self.gen_weights_path =self.model_save + '/' + type + '_gen.pth'
        self.dis_weights_path =self.model_save + '/' +  type + '_dis.pth'

        if os.path.exists(self.gen_weights_path):
            print('Loading %s generator...' % self.name)

            if torch.cuda.is_available():
                data = torch.load(self.gen_weights_path)
            else:
                data = torch.load(self.gen_weights_path, map_location=lambda storage, loc: storage)

            self.generator.load_state_dict(data['generator'])
            self.iteration = data['iteration']

        # load discriminator only when training
        if self.config.MODE == 1 and os.path.exists(self.dis_weights_path):
            print('Loading %s discriminator...' % self.name)

            if torch.cuda.is_available():
                data = torch.load(self.dis_weights_path)
            else:
                data = torch.load(self.dis_weights_path, map_location=lambda storage, loc: storage)

            self.discriminator.load_state_dict(data['discriminator'])

    def save(self):

        if len(self.config.GPU) > 1:
            generate_param = self.generator.module.state_dict()
            dis_param = self.discriminator.module.state_dict()
            print('save...multiple GPU')
        else:
            generate_param = self.generator.state_dict()
            dis_param = self.discriminator.state_dict()
            print('save...single GPU')

        torch.save({
            'iteration': self.iteration,
            'generator': generate_param
        }, os.path.join(self.model_save, '{}_{}_gen.pth'.format(self.iteration, self.name)))

        torch.save({
            'discriminator': dis_param
        }, os.path.join(self.model_save, '{}_{}_dis.pth'.format(self.iteration, self.name)))

        print('\nsaving %s...\n' % self.name)
class InpaintingModel(BaseModel):
    def __init__(self, config):
```

```
        super(InpaintingModel, self).__init__('InpaintingModel', config)

        generator = InpaintGenerator(config=config)
        discriminator = Discriminator(in_channels=3, use_sigmoid=config.GAN_LOSS != 'hinge')

        l1_loss = nn.L1Loss()
        perceptual_loss = PerceptualLoss()
        style_loss = StyleLoss()
        adversarial_loss = AdversarialLoss(type=config.GAN_LOSS)

        self.add_module('generator', generator)
        self.add_module('discriminator', discriminator)

        self.add_module('l1_loss', l1_loss)
        self.add_module('perceptual_loss', perceptual_loss)
        self.add_module('style_loss', style_loss)
        self.add_module('adversarial_loss', adversarial_loss)

        self.gen_optimizer = optim.Adam(
            params=generator.parameters(),
            lr=float(config.LR),
            betas=(config.BETA1, config.BETA2)
        )


        self.dis_optimizer = optim.Adam(
            params=discriminator.parameters(),
            lr=float(config.LR) * float(config.D2G_LR),
            betas=(config.BETA1, config.BETA2)
        )

    def process(self, images, masks):
        self.iteration += 1

        # zero optimizers
        self.gen_optimizer.zero_grad()
        self.dis_optimizer.zero_grad()


        # process outputs
        outputs = self(images, masks)
        gen_loss = 0
        dis_loss = 0


        # discriminator loss
        dis_input_real = images
        dis_input_fake = outputs.detach()
        dis_real, _ = self.discriminator(dis_input_real)                    # in: [rgb(3)]
        dis_fake, _ = self.discriminator(dis_input_fake)                    # in: [rgb(3)]
        dis_real_loss = self.adversarial_loss(dis_real, True, True)
        dis_fake_loss = self.adversarial_loss(dis_fake, False, True)
        dis_loss += (dis_real_loss + dis_fake_loss) / 2


        # generator adversarial loss
        gen_input_fake = outputs
        gen_fake, _ = self.discriminator(gen_input_fake)                    # in: [rgb(3)]
        gen_gan_loss = self.adversarial_loss(gen_fake, True, False) * self.config.INPAINT_ADV_LOSS_WEIGH
        gen_loss += gen_gan_loss


        # generator l1 loss
        gen_l1_loss = self.l1_loss(outputs, images) * self.config.L1_LOSS_WEIGHT / torch.mean(masks)
        gen_loss += gen_l1_loss


        # generator perceptual loss
        gen_content_loss = self.perceptual_loss(outputs, images)
        gen_content_loss = gen_content_loss * self.config.CONTENT_LOSS_WEIGHT
        gen_loss += gen_content_loss
        # generator style loss
```

```
    gen_style_loss = self.style_loss(outputs * masks, images * masks)
    gen_style_loss = gen_style_loss * self.config.STYLE_LOSS_WEIGHT
    gen_loss += gen_style_loss


    # create logs
    logs = [
        ("l_d2", dis_loss.item()),
        ("l_g2", gen_gan_loss.item()),
        ("l_l1", gen_l1_loss.item()),
        ("l_per", gen_content_loss.item()),
        ("l_sty", gen_style_loss.item()),
    ]

    return outputs, gen_loss, dis_loss, logs

def forward(self, images, masks):
    images_masked = images * (1 - masks)
    inputs = torch.cat((images_masked, masks), dim=1)
    outputs = self.generator(inputs)
    return outputs

def backward(self, gen_loss=None, dis_loss=None):
    gen_loss.backward()
    self.gen_optimizer.step()

    dis_loss.backward()
    self.dis_optimizer.step()
```

misf-main/src/config.py

```
import os
import yaml

class Config(dict):
    def __init__(self, config_path):
        with open(config_path, 'r') as f:
            self._yaml = f.read()
            self._dict = yaml.load(self._yaml)
            self._dict['PATH'] = os.path.dirname(config_path)

    def __getattr__(self, name):
        if self._dict.get(name) is not None:
            return self._dict[name]

        if DEFAULT_CONFIG.get(name) is not None:
            return DEFAULT_CONFIG[name]

        return None

    def print(self):
        print('Model configurations:')
        print('---------------------------------')
        print(self._yaml)
        print('')
        print('---------------------------------')
        print('')


DEFAULT_CONFIG = {
    'NMS': 1,
    'SEED': 10,                     # random seed
    'GPU': [0],                     # list of gpu ids
    'DEBUG': 0,                     # turns on debugging mode
    'VERBOSE': 0,                   # turns on verbose mode in the output console

    'LR': 0.0001,                   # learning rate
    'D2G_LR': 0.1,                  # discriminator/generator learning rate ratio
    'BETA1': 0.0,                   # adam optimizer beta1
    'BETA2': 0.9,                   # adam optimizer beta2
    'BATCH_SIZE': 8,                # input batch size for training
    'INPUT_SIZE': 256,              # input image size for training 0 for original size
    'MAX_ITERS': 2e6,               # maximum number of iterations to train the model

    'L1_LOSS_WEIGHT': 1,            # l1 loss weight
    'FM_LOSS_WEIGHT': 10,           # feature-matching loss weight
    'STYLE_LOSS_WEIGHT': 1,         # style loss weight
    'CONTENT_LOSS_WEIGHT': 1,       # perceptual loss weight
    'INPAINT_ADV_LOSS_WEIGHT': 0.01,# adversarial loss weight

    'GAN_LOSS': 'nsgan',            # nsgan | lsgan | hinge
    'GAN_POOL_SIZE': 0,             # fake images pool size

    'SAVE_INTERVAL': 1000,          # how many iterations to wait before saving model (0: never)
    'SAMPLE_INTERVAL': 1000,        # how many iterations to wait before sampling (0: never)
    'SAMPLE_SIZE': 12,              # number of images to sample
    'EVAL_INTERVAL': 0,             # how many iterations to wait before model evaluation (0: never)
    'LOG_INTERVAL': 10,             # how many iterations to wait before logging training status (0: nev
}
```

misf-main/src/utils.py

```
import os
import sys
import time
import random
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


def create_dir(dir):
    if not os.path.exists(dir):
        os.makedirs(dir)


def create_mask(width, height, mask_width, mask_height, x=None, y=None):
    mask = np.zeros((height, width))
    mask_x = x if x is not None else random.randint(0, width - mask_width)
    mask_y = y if y is not None else random.randint(0, height - mask_height)
    mask[mask_y:mask_y + mask_height, mask_x:mask_x + mask_width] = 1
    return mask


def stitch_images(inputs, *outputs, img_per_row=2):
    gap = 5
    columns = len(outputs) + 1

    width, height = inputs[0][:, :, 0].shape
    img = Image.new('RGB', (width * img_per_row * columns + gap * (img_per_row - 1), height * int(len(in
    images = [inputs, *outputs]

    for ix in range(len(inputs)):
        xoffset = int(ix % img_per_row) * width * columns + int(ix % img_per_row) * gap
        yoffset = int(ix / img_per_row) * height

        for cat in range(len(images)):
            im = np.array((images[cat][ix]).cpu()).astype(np.uint8).squeeze()
            im = Image.fromarray(im)
            img.paste(im, (xoffset + cat * width, yoffset))

    return img


def imshow(img, title=''):
    fig = plt.gcf()
    fig.canvas.set_window_title(title)
    plt.axis('off')
    plt.imshow(img, interpolation='none')
    plt.show()


def imsave(img, path):
    im = Image.fromarray(img.cpu().numpy().astype(np.uint8).squeeze())
    im.save(path)


class Progbar(object):

    def __init__(self, target, width=25, verbose=1, interval=0.05,
                 stateful_metrics=None):
        self.target = target
        self.width = width
        self.verbose = verbose
        self.interval = interval
        if stateful_metrics:
            self.stateful_metrics = set(stateful_metrics)
        else:
            self.stateful_metrics = set()
        self._dynamic_display = ((hasattr(sys.stdout, 'isatty') and
```

```
                             sys.stdout.isatty()) or
                            'ipykernel' in sys.modules or
                            'posix' in sys.modules)
        self._total_width = 0
        self._seen_so_far = 0
        # We use a dict + list to avoid garbage collection
        # issues found in OrderedDict
        self._values = {}
        self._values_order = []
        self._start = time.time()
        self._last_update = 0

    def update(self, current, values=None):

        values = values or []
        for k, v in values:
            if k not in self._values_order:
                self._values_order.append(k)
            if k not in self.stateful_metrics:
                if k not in self._values:
                    self._values[k] = [v * (current - self._seen_so_far),
                                       current - self._seen_so_far]
                else:
                    self._values[k][0] += v * (current - self._seen_so_far)
                    self._values[k][1] += (current - self._seen_so_far)
            else:
                self._values[k] = v
        self._seen_so_far = current

        now = time.time()
        info = ' - %.0fs' % (now - self._start)
        if self.verbose == 1:
            if (now - self._last_update < self.interval and
                    self.target is not None and current < self.target):
                return

            prev_total_width = self._total_width
            if self._dynamic_display:
                sys.stdout.write('\b' * prev_total_width)
                sys.stdout.write('\r')
            else:
                sys.stdout.write('\n')

            if self.target is not None:
                numdigits = int(np.floor(np.log10(self.target))) + 1
                barstr = '%%%dd/%d [' % (numdigits, self.target)
                bar = barstr % current
                prog = float(current) / self.target
                prog_width = int(self.width * prog)
                if prog_width > 0:
                    bar += ('=' * (prog_width - 1))
                    if current < self.target:
                        bar += '>'
                    else:
                        bar += '='
                bar += ('.' * (self.width - prog_width))
                bar += ']'
            else:
                bar = '%7d/Unknown' % current

            self._total_width = len(bar)
            sys.stdout.write(bar)

            if current:
                time_per_unit = (now - self._start) / current
            else:
                time_per_unit = 0
            if self.target is not None and current < self.target:
                eta = time_per_unit * (self.target - current)
                if eta > 3600:
                    eta_format = '%d:%02d:%02d' % (eta // 3600,
                                                   (eta % 3600) // 60,
```

```
                                                eta % 60)
                elif eta > 60:
                    eta_format = '%d:%02d' % (eta // 60, eta % 60)
                else:
                    eta_format = '%ds' % eta

                info = ' - ETA: %s' % eta_format
            else:
                if time_per_unit >= 1:
                    info += ' %.0fs/step' % time_per_unit
                elif time_per_unit >= 1e-3:
                    info += ' %.0fms/step' % (time_per_unit * 1e3)
                else:
                    info += ' %.0fus/step' % (time_per_unit * 1e6)

            for k in self._values_order:
                info += ' - %s:' % k
                if isinstance(self._values[k], list):
                    avg = np.mean(self._values[k][0] / max(1, self._values[k][1]))
                    if abs(avg) > 1e-3:
                        info += ' %.4f' % avg
                    else:
                        info += ' %.4e' % avg
                else:
                    info += ' %s' % self._values[k]

            self._total_width += len(info)
            if prev_total_width > self._total_width:
                info += (' ' * (prev_total_width - self._total_width))

            if self.target is not None and current >= self.target:
                info += '\n'

            sys.stdout.write(info)
            sys.stdout.flush()

        elif self.verbose == 2:
            if self.target is None or current >= self.target:
                for k in self._values_order:
                    info += ' - %s:' % k
                    avg = np.mean(self._values[k][0] / max(1, self._values[k][1]))
                    if avg > 1e-3:
                        info += ' %.4f' % avg
                    else:
                        info += ' %.4e' % avg
                info += '\n'

                sys.stdout.write(info)
                sys.stdout.flush()

        self._last_update = now

    def add(self, n, values=None):
        self.update(self._seen_so_far + n, values)
```

misf-main/src/misf.py

```
import os
import numpy as np
import torch
from torch.utils.data import DataLoader
from .dataset import Dataset
from .models import InpaintingModel
from .utils import Progbar, create_dir
from .metrics import PSNR
from skimage.metrics import structural_similarity as compare_ssim
from skimage.metrics import peak_signal_noise_ratio as compare_psnr
import kpn.utils as kpn_utils
import torchvision
import lpips


class MISF():
    def __init__(self, config):
        self.config = config

        self.debug = False
        self.inpaint_model = InpaintingModel(config).to(config.DEVICE)

        self.transf = torchvision.transforms.Compose(
            [
                torchvision.transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])])
        self.loss_fn_vgg = lpips.LPIPS(net='vgg').to(config.DEVICE)


        self.psnr = PSNR(255.0).to(config.DEVICE)

        # test mode
        if self.config.MODE == 2:
            self.test_dataset = Dataset(config, config.TEST_FLIST, config.TEST_MASK_FLIST, augment=False
            print('test dataset:'.format(len(self.test_dataset)))
        else:
            self.train_dataset = Dataset(config, config.TRAIN_FLIST, config.TRAIN_MASK_FLIST, augment=Tr
            self.val_dataset = Dataset(config, config.VAL_FLIST, config.VAL_MASK_FLIST, augment=False, t
            self.sample_iterator = self.val_dataset.create_iterator(config.SAMPLE_SIZE)

            print('train dataset:{}'.format(len(self.train_dataset)))
            print('eval dataset:{}'.format(len(self.val_dataset)))

        self.samples_path = os.path.join(config.PATH, 'samples')
        self.results_path = os.path.join(config.PATH, 'results')

        if config.RESULTS is not None:
            self.results_path = os.path.join(config.RESULTS)

        if config.DEBUG is not None and config.DEBUG != 0:
            self.debug = True


    def load(self):
        self.inpaint_model.load(self.config.MODEL_LOAD)

    def save(self):
        self.inpaint_model.save()

    def train(self):
        train_loader = DataLoader(
            dataset=self.train_dataset,
            batch_size=self.config.BATCH_SIZE,
            num_workers=0,
            drop_last=True,
            shuffle=True
        )

        epoch = 0
```

```
        keep_training = True
        max_iteration = int(float((self.config.MAX_ITERS)))
        total = len(self.train_dataset)

        if total == 0:
            print('No training data was provided! Check \'TRAIN_FLIST\' value in the configuration file.
            return

        max_psnr = 0
        while(keep_training):
            epoch += 1
            print('\n\nTraining epoch: %d' % epoch)

            for items in train_loader:
                self.inpaint_model.train()

                images, masks = self.cuda(*items)

                outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, masks)
                outputs_merged = (outputs * masks) + images * (1 - masks)

                # backward
                self.inpaint_model.backward(gen_loss, dis_loss)
                iteration = self.inpaint_model.iteration

                if iteration >= max_iteration:
                    keep_training = False
                    break

                logs = [
                    ("epoch", epoch),
                    ("iter", iteration),
                ] + logs


                # sample
                if iteration % self.config.TRAIN_SAMPLE_INTERVAL == 0:
                    img_list2 = [images * (1 - masks), outputs_merged, outputs, images]
                    name_list2 = ['in', 'pred_2', 'pre_1', 'gt']
                    kpn_utils.save_sample_png(sample_folder=self.config.TRAIN_SAMPLE_SAVE,
                                              sample_name='ite_{}_{}'.format(self.inpaint_model.iteratio
                                                                             0), img_list=img_list2,
                                              name_list=name_list2, pixel_max_cnt=255, height=-1,
                                              width=-1)


                # save model at checkpoints
                if iteration % self.config.SAVE_INTERVAL == 0:
                    self.save()

                # evaluate model at checkpoints
                if iteration % self.config.EVAL_INTERVAL == 0:
                    print('\nstart eval...\n')
                    cur_psnr = self.eval()
                    self.inpaint_model.iteration = iteration

                    if cur_psnr > max_psnr:
                        max_psnr = cur_psnr
                        self.save()
                        print('---increase-iteration:{}'.format(iteration))

                print(logs)

        print('\nEnd training....')

    def eval(self):
        val_loader = DataLoader(
            dataset=self.val_dataset,
            batch_size=1,
            drop_last=True,
            shuffle=False
        )
```

```
        model = self.config.MODEL

        self.inpaint_model.eval()

        psnr_all = []
        ssim_all = []
        l1_list = []
        lpips_list = []

        iteration = self.inpaint_model.iteration
        with torch.no_grad():
            for items in val_loader:
                images, masks = self.cuda(*items)

                outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, masks)
                outputs_merged = (outputs * masks) + images * (1 - masks)

                psnr, ssim = self.metric(images, outputs_merged)
                psnr_all.append(psnr)
                ssim_all.append(ssim)

                l1_loss = torch.nn.functional.l1_loss(outputs_merged, images, reduction='mean').item()
                l1_list.append(l1_loss)

                pl = 1.0
                lpips_list.append(pl)
                # if torch.cuda.is_available():
                #     pl = loss_fn_vgg(transf(outputs_merged[0].cpu()).cuda(), transf(images[0].cpu()).c
                #     lpips_list.append(pl)
                # else:
                #     pl = loss_fn_vgg(transf(outputs_merged[0].cpu()), transf(images[0].cpu())).item()
                #     lpips_list.append(pl)


                # sample
                if len(psnr_all) % self.config.EVAL_SAMPLE_INTERVAL == 0:
                    img_list2 = [images * (1 - masks), outputs_merged, outputs, images]
                    name_list2 = ['in', 'pred2', 'pre1', 'gt']
                    kpn_utils.save_sample_png(sample_folder=self.config.EVAL_SAMPLE_SAVE,
                                              sample_name='ite_{}_{}'.format(iteration, len(psnr_all)), img_
                                              name_list=name_list2, pixel_max_cnt=255, height=-1,
                                              width=-1)



                print('psnr:{}/{}  ssim:{}/{} l1:{}/{}  lpips{}/{}  {}/{}'.format(psnr, np.average(psnr_
                                                                                  ssim, np.average(ssim
                                                                                  l1_loss, np.average(l
                                                                                  pl, np.average(lpips_
                                                                                  len(psnr_all), len(se

                if len(psnr_all) >= 1000:
                    break

            print('iteration:{} ave_psnr:{}  ave_ssim:{} ave_l1:{}  ave_lpips:{}'.format(
                iteration,
                np.average(psnr_all),
                np.average(ssim_all),
                np.average(l1_list),
                np.average(lpips_list)
            ))

            return np.average(psnr_all)

    def test(self):
        self.inpaint_model.eval()

        create_dir(self.results_path)

        test_loader = DataLoader(
            dataset=self.test_dataset,
            batch_size=1,
```

```
    )

    psnr_list = []
    ssim_list = []
    l1_list = []
    lpips_list = []

    index = 0
    with torch.no_grad():
        for items in test_loader:
            images, masks = self.cuda(*items)
            index += 1

            outputs = self.inpaint_model(images, masks)
            outputs_merged = (outputs * masks) + (images * (1 - masks))

            psnr, ssim = self.metric(images, outputs_merged)
            psnr_list.append(psnr)
            ssim_list.append(ssim)

            if torch.cuda.is_available():
                pl = self.loss_fn_vgg(self.transf(outputs_merged[0].cpu()).cuda(), self.transf(image
                lpips_list.append(pl)
            else:
                pl = self.loss_fn_vgg(self.transf(outputs_merged[0].cpu()), self.transf(images[0].cp
                lpips_list.append(pl)

            l1_loss = torch.nn.functional.l1_loss(outputs_merged, images, reduction='mean').item()
            l1_list.append(l1_loss)

            print("psnr:{}/{}  ssim:{}/{} l1:{}/{}  lpips:{}/{}  {}".format(psnr, np.average(psnr_li
                                                                            ssim, np.average(ssim_li
                                                                            l1_loss, np.average(l1_l
                                                                            pl, np.average(lpips_lis
                                                                            len(ssim_list)))


            if len(ssim_list) % 1 == 0:
                images_masked = images * (1 - masks)
                img_list = [images_masked, images, outputs, outputs_merged]
                name_list = ['in', 'gt', 'pre1', 'pre2']

                kpn_utils.save_sample_png(sample_folder=self.config.TEST_SAMPLE_SAVE, sample_name='{
                                          img_list=img_list,
                                          name_list=name_list, pixel_max_cnt=255, height=-1, width=-

        print('psnr_ave:{} ssim_ave:{} l1_ave:{} lpips:{}'.format(np.average(psnr_list),
                                                                  np.average(ssim_list),
                                                                  np.average(l1_list),
                                                                  np.average(lpips_list))


def cuda(self, *args):
    return (item.to(self.config.DEVICE) for item in args)

def postprocess(self, img):
    # [0, 1] => [0, 255]
    img = img * 255.0
    img = img.permute(0, 2, 3, 1)
    return img.int()

def metric(self, gt, pre):
    pre = pre.clamp_(0, 1) * 255.0
    pre = pre.permute(0, 2, 3, 1)
    pre = pre.detach().cpu().numpy().astype(np.uint8)[0]

    gt = gt.clamp_(0, 1) * 255.0
    gt = gt.permute(0, 2, 3, 1)
    gt = gt.cpu().detach().numpy().astype(np.uint8)[0]

    psnr = min(100, compare_psnr(gt, pre))
    ssim = compare_ssim(gt, pre, multichannel=True, data_range=255)
```

```
return psnr, ssim
```

misf-main/src/loss.py

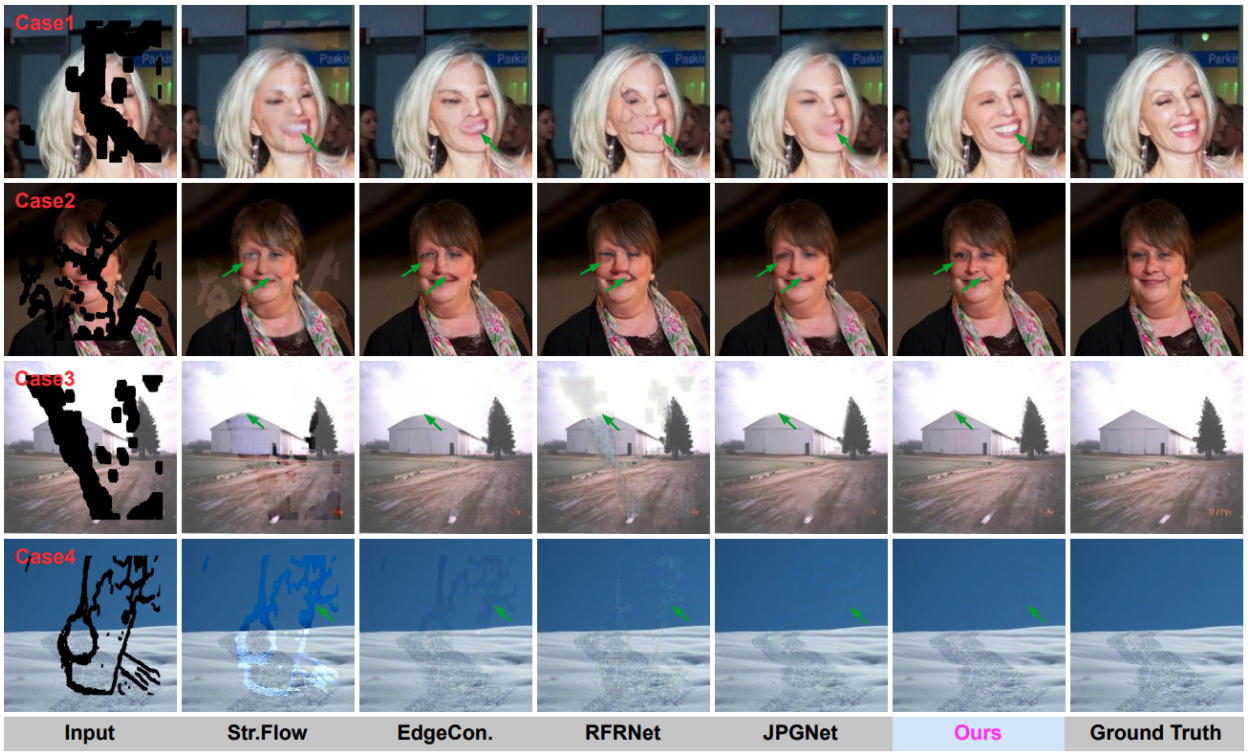
```
import torch
import torch.nn as nn
import torchvision.models as models


class AdversarialLoss(nn.Module):


    def __init__(self, type='nsgan', target_real_label=1.0, target_fake_label=0.0):
        r"""
        type = nsgan | lsgan | hinge
        """
        super(AdversarialLoss, self).__init__()

        self.type = type
        self.register_buffer('real_label', torch.tensor(target_real_label))
        self.register_buffer('fake_label', torch.tensor(target_fake_label))

        if type == 'nsgan':
            self.criterion = nn.BCELoss()

        elif type == 'lsgan':
            self.criterion = nn.MSELoss()

        elif type == 'hinge':
            self.criterion = nn.ReLU()

    def __call__(self, outputs, is_real, is_disc=None):
        if self.type == 'hinge':
            if is_disc:
                if is_real:
                    outputs = -outputs
                return self.criterion(1 + outputs).mean()
            else:
                return (-outputs).mean()

        else:
            labels = (self.real_label if is_real else self.fake_label).expand_as(outputs)
            loss = self.criterion(outputs, labels)
            return loss


class StyleLoss(nn.Module):


    def __init__(self):
        super(StyleLoss, self).__init__()
        self.add_module('vgg', VGG19())
        self.criterion = torch.nn.L1Loss()

    def compute_gram(self, x):
        b, ch, h, w = x.size()
        f = x.view(b, ch, w * h)
        f_T = f.transpose(1, 2)
        G = f.bmm(f_T) / (h * w * ch)

        return G

    def __call__(self, x, y):
        # Compute features
        x_vgg, y_vgg = self.vgg(x), self.vgg(y)

        # Compute loss
        style_loss = 0.0
        style_loss += self.criterion(self.compute_gram(x_vgg['relu2_2']), self.compute_gram(y_vgg['relu2
        style_loss += self.criterion(self.compute_gram(x_vgg['relu3_4']), self.compute_gram(y_vgg['relu3
        style_loss += self.criterion(self.compute_gram(x_vgg['relu4_4']), self.compute_gram(y_vgg['relu4
        style_loss += self.criterion(self.compute_gram(x_vgg['relu5_2']), self.compute_gram(y_vgg['relu5
```

```
        return style_loss


class PerceptualLoss(nn.Module):


    def __init__(self, weights=[1.0, 1.0, 1.0, 1.0, 1.0]):
        super(PerceptualLoss, self).__init__()
        self.add_module('vgg', VGG19())
        self.criterion = torch.nn.L1Loss()
        self.weights = weights

    def __call__(self, x, y):
        # Compute features
        x_vgg, y_vgg = self.vgg(x), self.vgg(y)

        content_loss = 0.0
        content_loss += self.weights[0] * self.criterion(x_vgg['relu1_1'], y_vgg['relu1_1'])
        content_loss += self.weights[1] * self.criterion(x_vgg['relu2_1'], y_vgg['relu2_1'])
        content_loss += self.weights[2] * self.criterion(x_vgg['relu3_1'], y_vgg['relu3_1'])
        content_loss += self.weights[3] * self.criterion(x_vgg['relu4_1'], y_vgg['relu4_1'])
        content_loss += self.weights[4] * self.criterion(x_vgg['relu5_1'], y_vgg['relu5_1'])


        return content_loss



class VGG19(torch.nn.Module):
    def __init__(self):
        super(VGG19, self).__init__()
        features = models.vgg19(pretrained=True).features
        self.relu1_1 = torch.nn.Sequential()
        self.relu1_2 = torch.nn.Sequential()

        self.relu2_1 = torch.nn.Sequential()
        self.relu2_2 = torch.nn.Sequential()

        self.relu3_1 = torch.nn.Sequential()
        self.relu3_2 = torch.nn.Sequential()
        self.relu3_3 = torch.nn.Sequential()
        self.relu3_4 = torch.nn.Sequential()

        self.relu4_1 = torch.nn.Sequential()
        self.relu4_2 = torch.nn.Sequential()
        self.relu4_3 = torch.nn.Sequential()
        self.relu4_4 = torch.nn.Sequential()

        self.relu5_1 = torch.nn.Sequential()
        self.relu5_2 = torch.nn.Sequential()
        self.relu5_3 = torch.nn.Sequential()
        self.relu5_4 = torch.nn.Sequential()

        for x in range(2):
            self.relu1_1.add_module(str(x), features[x])

        for x in range(2, 4):
            self.relu1_2.add_module(str(x), features[x])

        for x in range(4, 7):
            self.relu2_1.add_module(str(x), features[x])

        for x in range(7, 9):
            self.relu2_2.add_module(str(x), features[x])

        for x in range(9, 12):
            self.relu3_1.add_module(str(x), features[x])

        for x in range(12, 14):
            self.relu3_2.add_module(str(x), features[x])
        for x in range(14, 16):
```

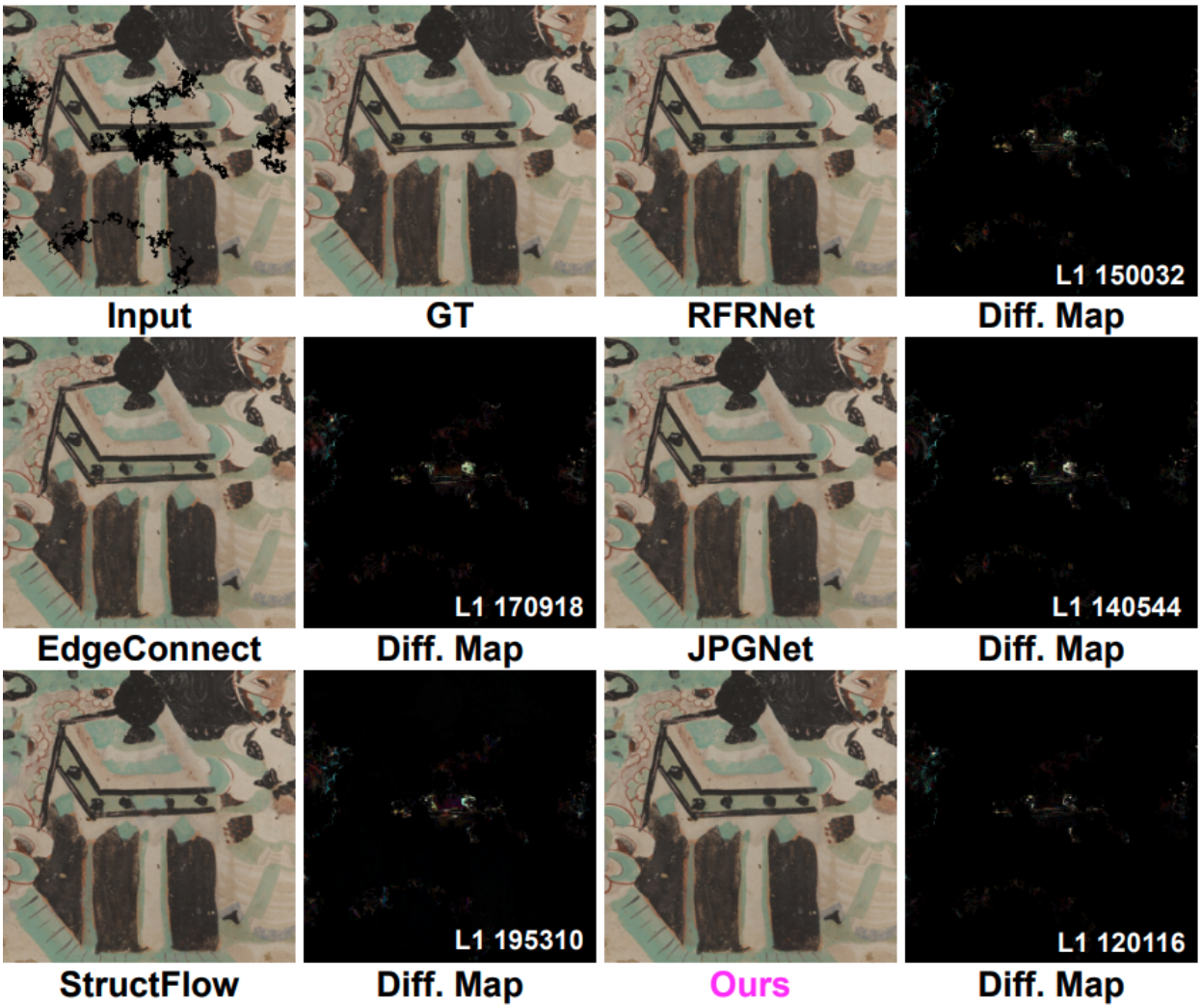
```
            self.relu3_3.add_module(str(x), features[x])

        for x in range(16, 18):
            self.relu3_4.add_module(str(x), features[x])

        for x in range(18, 21):
            self.relu4_1.add_module(str(x), features[x])

        for x in range(21, 23):
            self.relu4_2.add_module(str(x), features[x])

        for x in range(23, 25):
            self.relu4_3.add_module(str(x), features[x])

        for x in range(25, 27):
            self.relu4_4.add_module(str(x), features[x])

        for x in range(27, 30):
            self.relu5_1.add_module(str(x), features[x])

        for x in range(30, 32):
            self.relu5_2.add_module(str(x), features[x])

        for x in range(32, 34):
            self.relu5_3.add_module(str(x), features[x])

        for x in range(34, 36):
            self.relu5_4.add_module(str(x), features[x])

        # don't need the gradients, just want the features
        for param in self.parameters():
            param.requires_grad = False

    def forward(self, x):
        relu1_1 = self.relu1_1(x)
        relu1_2 = self.relu1_2(relu1_1)

        relu2_1 = self.relu2_1(relu1_2)
        relu2_2 = self.relu2_2(relu2_1)

        relu3_1 = self.relu3_1(relu2_2)
        relu3_2 = self.relu3_2(relu3_1)
        relu3_3 = self.relu3_3(relu3_2)
        relu3_4 = self.relu3_4(relu3_3)

        relu4_1 = self.relu4_1(relu3_4)
        relu4_2 = self.relu4_2(relu4_1)
        relu4_3 = self.relu4_3(relu4_2)
        relu4_4 = self.relu4_4(relu4_3)

        relu5_1 = self.relu5_1(relu4_4)
        relu5_2 = self.relu5_2(relu5_1)
        relu5_3 = self.relu5_3(relu5_2)
        relu5_4 = self.relu5_4(relu5_3)

        out = {
            'relu1_1': relu1_1,
            'relu1_2': relu1_2,

            'relu2_1': relu2_1,
            'relu2_2': relu2_2,

            'relu3_1': relu3_1,
            'relu3_2': relu3_2,
            'relu3_3': relu3_3,
            'relu3_4': relu3_4,

            'relu4_1': relu4_1,
            'relu4_2': relu4_2,
            'relu4_3': relu4_3,
            'relu4_4': relu4_4,
            'relu5_1': relu5_1,
```

```
    'relu5_2': relu5_2,
    'relu5_3': relu5_3,
    'relu5_4': relu5_4,
}
return out
```

misf-main/images/comparison.png



misf-main/images/dunhuang_diff.png

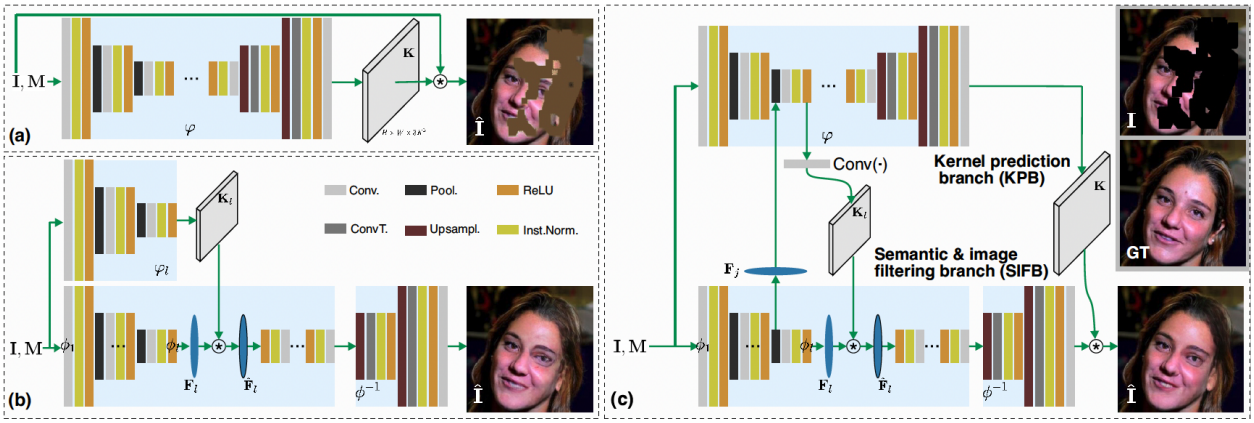


misf-main/images/misf_arch.png

Encoder-decoder ($\phi^{-1}(\phi(\cdot))$)				Predictive network ($\varphi(\cdot)$)			
In.	Out.	Out. size	Layers	In.	Out.	Out. size	Layers
$\mathbf{I}$	$\mathbf{F}_1$	256×256	conv(7, 3, 64)	$\mathbf{I}$	$\mathbf{E}_1$	256×256	conv(7, 3, 64)
$\mathbf{F}_1$	$\mathbf{F}_2$	128×128	conv(4, 64, 128)	$\mathbf{E}_1$	$\mathbf{E}_2$	128×128	conv(4, 64, 128)
$\mathbf{F}_2$	$\mathbf{F}'_2$	64×64	AvgPool	$\mathbf{E}_2$	$\mathbf{E}'_2$	64×64	AvgPool
$\mathbf{F}'_2$	$\mathbf{F}_3$	64×64	conv(4, 128, 256)	$[\mathbf{F}'_2, \mathbf{E}'_2]$	$\mathbf{E}_3$	64×64	conv(4, 256, 256)
$\mathbf{F}_3$	$\hat{\mathbf{F}}_3$	64×64	$\mathbf{F}_3 \circledast \mathbf{K}_3$	$\mathbf{E}_3$	$\mathbf{K}_3$	$64^2 \times 256N^2$	Conv(1, 256, $256N^2$)
$\hat{\mathbf{F}}_3$	$\mathbf{F}_4$	64×64	$8\times$ conv(1, 256, 256)	$\mathbf{E}_3$	$\mathbf{E}_4$	64×64	$8\times$ conv(1, 256, 256)
$\mathbf{F}_4$	$\mathbf{F}_5$	64×64	convt(4, 256, 128)	$\mathbf{E}_4$	$\mathbf{E}_5$	64×64	convt(4, 256, 128)
$\mathbf{F}_5$	$\mathbf{F}_6$	128×128	convt(4, 128, 64)	$\mathbf{E}_5$	$\mathbf{E}_6$	128×128	convt(4, 128, 64)
$\mathbf{F}_6$	$\mathbf{F}_7$	256×256	convt(7, 64, C)	$\mathbf{E}_6$	$\mathbf{K}$	256×256	convt(7, 64, CN^2)
$\mathbf{F}_7$	$\hat{\mathbf{I}}$	256×256	$\mathbf{F}_7 \circledast \mathbf{K}$	-	-	-	-

Table 1. Architecture of MISF. Network architectures of the encoder-decoder network (*i.e.*, $\phi(\cdot)$ and$\phi^{-1}(\cdot)$) and the predictive network (*i.e.*, $\varphi(\cdot)$). The variable N is the kernel size of filtering, *i.e.*, $|\mathcal{N}_p| = N^2$, and $C = 3$ denotes the number of color channel. 'conv(x,x,x)' defines the kernel size, input and output channel numbers, respectively.

misf-main/images/frameworks.png



misf-main/data/mask.txt

["/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/mask/04024.png", "/Users/xi

misf-main/data/face.txt

["/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/face/182649.jpg", "/Users/x

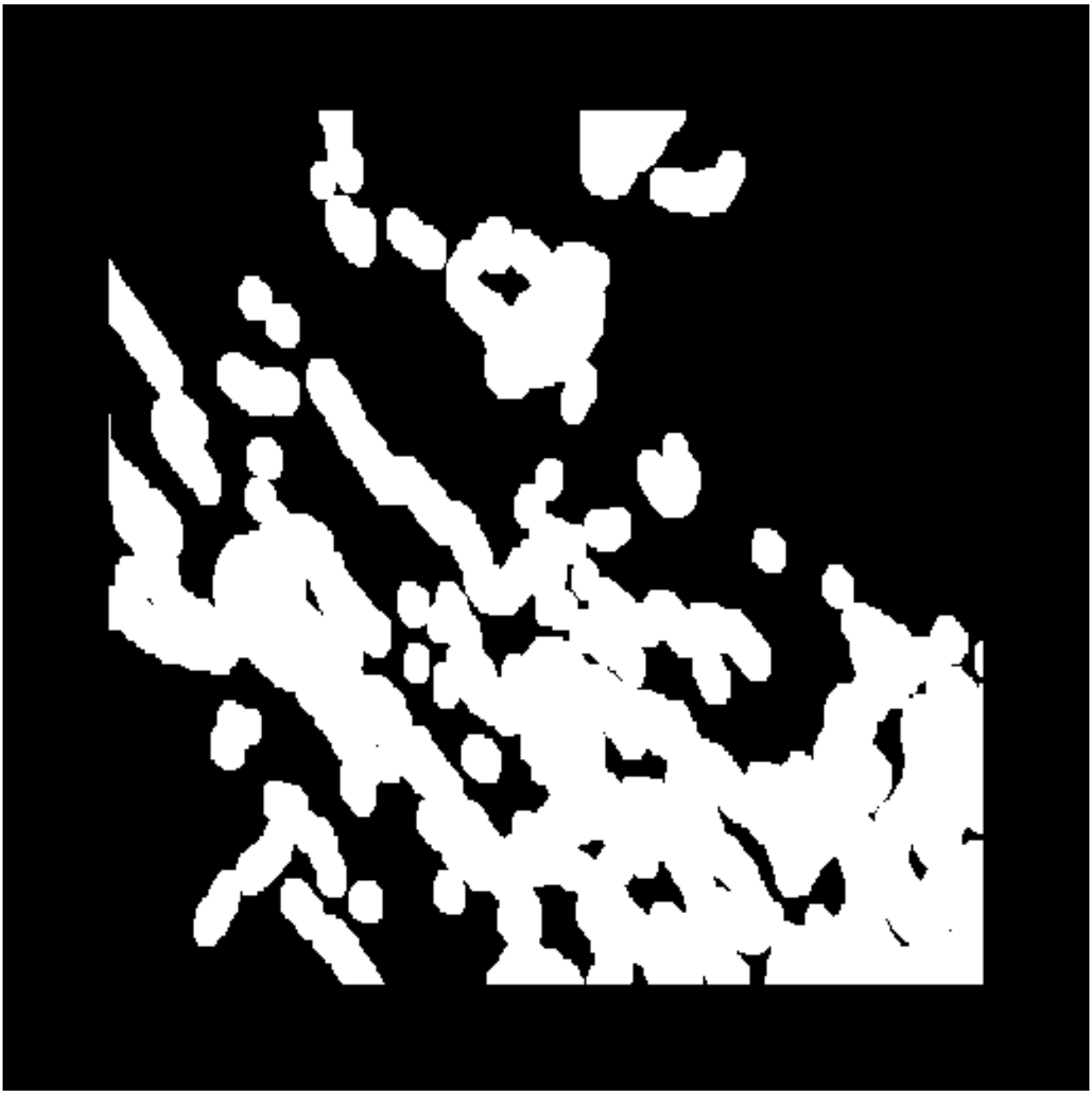
misf-main/data/data_list.py

```
import json
import os
def load_file_list_recursion(fpath, result):
    allfilelist = os.listdir(fpath)
    for file in allfilelist:
        filepath = os.path.join(fpath, file)
        if os.path.isdir(filepath):
            load_file_list_recursion(filepath, result)
        else:
            result.append(filepath)
            print(len(result))



def scan(input_path, out_put):
    result_list = []
    load_file_list_recursion(input_path, result_list)
    result_list.sort()

    for i in range(len(result_list)):
        print('{}_{}'.format(i, result_list[i]))

    with open(out_put, 'w') as j:
        json.dump(result_list, j)

scan('/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/face', './face.txt')
scan('/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/mask', './mask.txt')
```

misf-main/data/mask/04024.png

misf-main/data/mask/04045.png

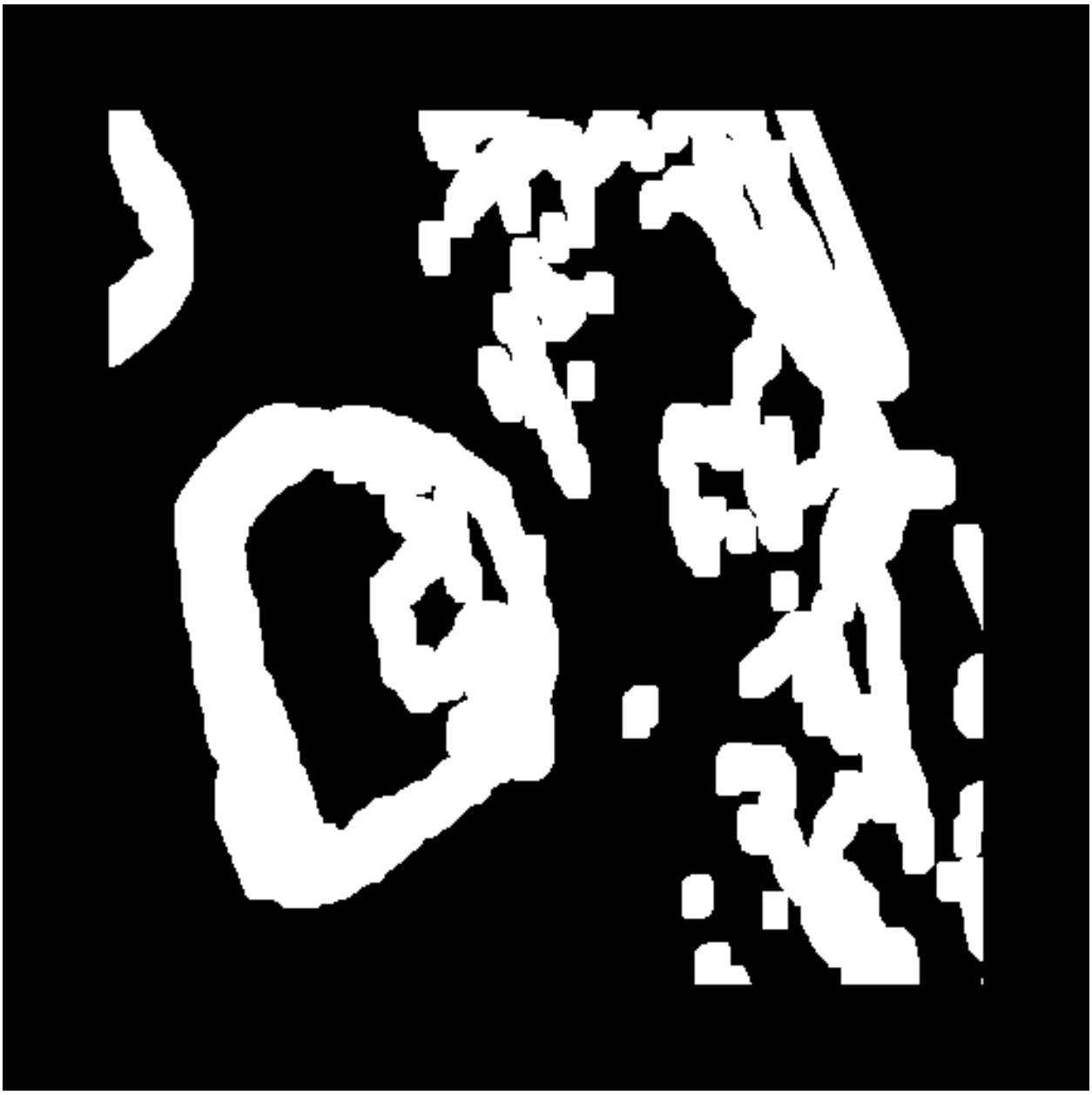
misf-main/data/mask/04051.png



misf-main/data/data_list.py

```
import json
import os
def load_file_list_recursion(fpath, result):
    allfilelist = os.listdir(fpath)
    for file in allfilelist:
        filepath = os.path.join(fpath, file)
        if os.path.isdir(filepath):
            load_file_list_recursion(filepath, result)
        else:
            result.append(filepath)
            print(len(result))



def scan(input_path, out_put):
    result_list = []
    load_file_list_recursion(input_path, result_list)
    result_list.sort()

    for i in range(len(result_list)):
        print('{}_{}'.format(i, result_list[i]))

    with open(out_put, 'w') as j:
        json.dump(result_list, j)

scan('/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/face', './face.txt')
scan('/Users/xiaoguangli/lxg/CV/publication/cvpr_2022/publication/code/misf/data/mask', './mask.txt')
```

misf-main/data/mask/04030.png



misf-main/data/face/182669.jpg

misf-main/data/face/182649.jpg

misf-main/data/face/182660.jpg

misf-main/data/face/182650.jpg

misf-main/data/face/182681.jpg

misf-main/test.py

```
from main import main
main(mode=2)
```