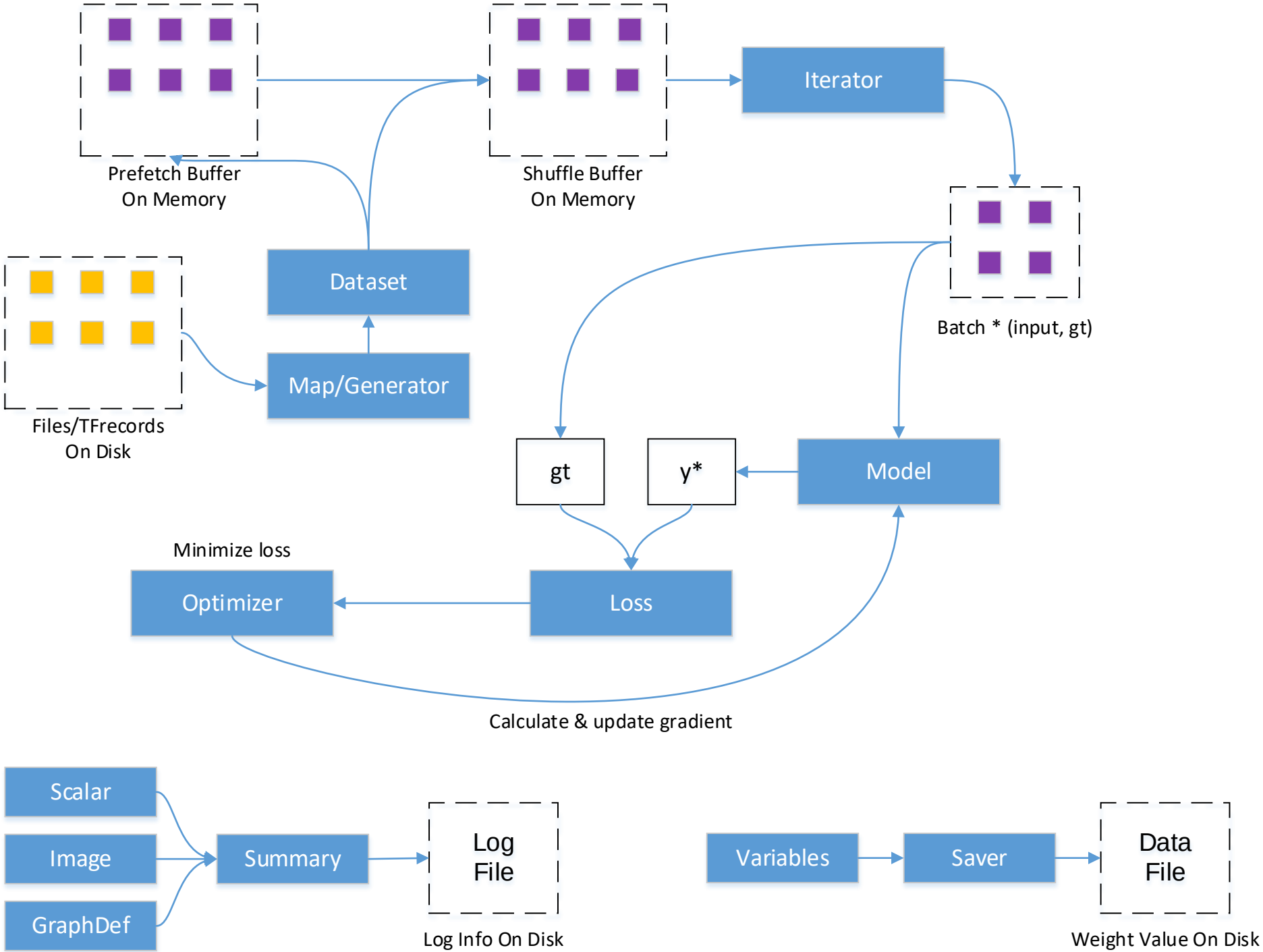
训练数据读取

TensorFlow

林楚铭

cmlin17@fudan.edu.cn

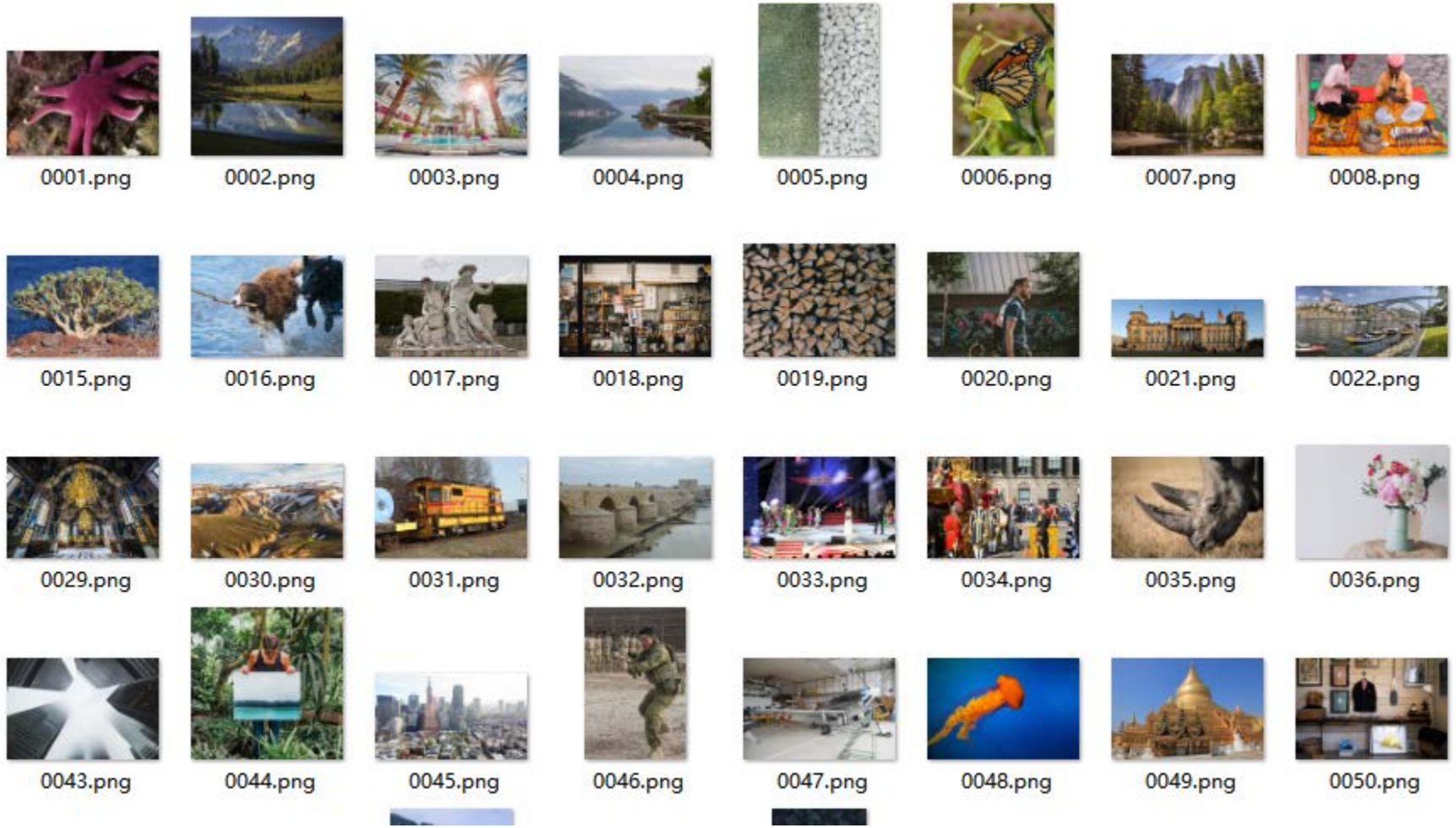

Prefetch Buffer
On Memory
Shuffle Buffer
On Memory
Iterator
Batch * (input, gt)
Dataset
Map/Generator
Files/TFrecords
On Disk
gt
y*
Model
Minimize loss
Optimizer
Loss
Calculate & update gradient
Scalar
Image
GraphDef
Summary
Log
File
Log Info On Disk
Variables
Saver
Data
File
Weight Value On Disk

目的

- 假设4倍超分模型为F
- 监督学习 HR_P = F(LR)
- 需要准备训练数据(LR, HR_GT)

- 训练数据采用DIV2K，一共800张图片
- LR为24x24，HR为96x96
- 生成训练样本：LR [B, 24, 24, 1] 与对应的 HR [B, 96, 96, 1]

DIV2K


0001.png
0002.png
0003.png
0004.png
0005.png
0006.png
0007.png
0008.png
0015.png
0016.png
0017.png
0018.png
0019.png
0020.png
0021.png
0022.png
0029.png
0030.png
0031.png
0032.png
0033.png
0034.png
0035.png
0036.png
0043.png
0044.png
0045.png
0046.png
0047.png
0048.png
0049.png
0050.png

简介

- 使用tf.data
- 两种读取数据模式：
  - 将原始数据处理成tfrecords数据，再从tfrecords读取
  - 使用生成器generator，直接处理原始数据

定义DataLoader

```python
class DataLoader():
    def __init__(self, data_dir='DIV2K/DIV2K_train_HR',
                 patch_size=96, scale=4, batch_size=64,
                 shuffle_num=2000, prefetch_num=1000,
                 map_parallel_num=8, one_img_patch_num=100):
        self.data_dir = data_dir
        self.patch_size = patch_size
        self.scale = scale
        self.batch_size = batch_size
        self.shuffle_num = shuffle_num
        self.prefetch_num = prefetch_num
        self.map_parallel_num = map_parallel_num
        self.one_img_patch_num = one_img_patch_num
```

生成tfrecords

```python
def gen_tfrecords(self, save_dir='DIV2K/tfrecords', tfrecord_num=10):
    file_num = tfrecord_num
    sample_num = 0
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    fs = []
    for i in range(file_num):
        fs.append(tf.python_io.TFRecordWriter(os.path.join(save_dir, 'data%d.tfrecords' % i)))

    img_paths = sorted(glob.glob(os.path.join(self.data_dir, '*.png')))
    for img_path in img_paths:
        print('processing %s' % img_path)
        img = misc.imread(img_path)
        y = utils.img_to_uint8(utils.rgb2ycbcr(img)[:, :, 0])
        height, width = y.shape
        p = self.patch_size
        step = p // 3 * 2
        for h in range(0, height - p + 1, step):
            for w in range(0, width - p + 1, step):
                gt = y[h: h + p, w: w + p]
                assert gt.shape[:] == (p, p)
                assert gt.dtype == np.uint8
                example = tf.train.Example(features=tf.train.Features(feature={
                    'gt': _bytes_feature(gt.tostring())
                }))
                fs[sample_num % file_num].write(example.SerializeToString())
                sample_num += 1
    print('example number: %d' % sample_num)
    np.savetxt(os.path.join(save_dir, 'sample_num.txt'), np.asarray([sample_num]), '%d')
    for f in fs:
        f.close()
```

为什么要生成10个tfrecords?

使用tf.data读取tfrecords

```python
def _parse_one_example(self, example):
    features = tf.parse_single_example(
        example,
        features={
            'gt': tf.FixedLenFeature([], tf.string)
        })
    p = self.patch_size
    gt = features['gt']
    gt = tf.decode_raw(gt, tf.uint8)
    gt = tf.reshape(gt, [p, p])
    gt = tf.cast(gt, tf.float32)

    lr = tf.py_func(lambda x: misc.imresize(x, 1.0 / self.scale, 'bicubic', 'F'), [gt], tf.float32)

    gt = tf.reshape(gt, [p, p, 1]) / 255.0
    lr = tf.reshape(lr, [p // self.scale, p // self.scale, 1]) / 255.0

    c1 = tf.random_uniform([], 0, 1)
    c2 = tf.random_uniform([], 0, 1)
    gt, lr = tf.cond(c1 < 0.5, lambda : (gt[::-1, :], lr[::-1, :]), lambda : (gt, lr))
    gt, lr = tf.cond(c2 < 0.5, lambda: (gt[:, ::-1], lr[:, ::-1]), lambda: (gt, lr))
    return lr, gt

def read_tfrecords(self, save_dir='DIV2K/tfrecords'):
    fs_paths = sorted(glob.glob(os.path.join(save_dir, '*.tfrecords')))
    dataset = tf.data.TFRecordDataset(fs_paths)
    dataset = dataset.map(self._parse_one_example, self.map_parallel_num).shuffle(self.shuffle_num) \
        .prefetch(self.prefetch_num).batch(self.batch_size).repeat()
    lrs, gts = dataset.make_one_shot_iterator().get_next()
    return lrs, gts
```

为什么需要tf.py_func?
优点与缺点？

为什么需要shuffle与prefetch?
Shuffle的大小如何决策？
map/shuffle/batch/repeat调用顺序是否可以交换？

使用tf.data与生成器读取数据

```python
def get_generator(self):
    img_paths = sorted(glob.glob(os.path.join(self.data_dir, '*.png')))
    one_img_patch_num = self.one_img_patch_num
    p = self.patch_size
    scale = self.scale
    for img_path in img_paths:
        img = misc.imread(img_path)
        height, width, _ = img.shape
        for i in range(one_img_patch_num):
            h = np.random.randint(height - p + 1)
            w = np.random.randint(width - p + 1)
            patch = img[h: h + p, w: w + p]
            gt = utils.rgb2ycbcr(patch)[:, :, 0]
            gt = np.float32(gt)
            lr = misc.imresize(gt, 1.0 / scale, 'bicubic', 'F')
            gt = gt / 255.0
            lr = lr / 255.0
            c1 = np.random.rand()
            c2 = np.random.rand()
            if c1 < 0.5:
                gt, lr = gt[::-1, :], lr[::-1, :]
            if c2 < 0.5:
                gt, lr = gt[:, ::-1], lr[:, ::-1]
            yield lr, gt

def read_pngs(self):
    dataset = tf.data.Dataset.from_generator(self.get_generator, (tf.float32, tf.float32))
    dataset = dataset.shuffle(self.shuffle_num).prefetch(self.prefetch_num).batch(self.batch_size).repeat()
    lrs, gts = dataset.make_one_shot_iterator().get_next()
    p = self.patch_size
    lrs = tf.reshape(lrs, [-1, p // self.scale, p // self.scale, 1])
    gts = tf.reshape(gts, [-1, p, p, 1])
    return lrs, gts
```

使用tfrecords读取与generator读取数据各有什么优点？两者分别的应用场景？

测试

```
if __name__ == '__main__':
    os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    data_loader = DataLoader()
    # data_loader.gen_tfrecords()
    # lrs, gts = data_loader.read_tfrecords()
    lrs, gts = data_loader.read_pngs()
    sess = tf.Session()
    # import matplotlib.pyplot as plt
    # while True:
    #     im1, im2 = sess.run([lrs, gts])
    #     plt.imshow(utils.img_to_uint8(im2[0, :, :, 0]))
    #     plt.show()
    im1, im2 = sess.run([lrs, gts])
    print(im1.shape, im2.shape)

Out: (64, 24, 24, 1) (64, 96, 96, 1)
```

两者性能对比

- 左：TFrecords　右：Generator

```
[0:00:06.458232] Step:360, loss:2.1939547061920166
[0:00:06.360591] Step:380, loss:1.7203850746154785
[0:00:06.497405] Step:400, loss:1.697927474975586
[0:00:06.392951] Step:420, loss:1.917050838470459
[0:00:06.406135] Step:440, loss:1.6480821371078491
[0:00:06.414975] Step:460, loss:2.2578155994415283
[0:00:06.417290] Step:480, loss:2.329984664916992
[0:00:06.418469] Step:500, loss:2.050870895385742
[0:00:06.424043] Step:520, loss:2.2898924350738525
[0:00:06.477182] Step:540, loss:1.6806976795196533
[0:00:06.477095] Step:560, loss:2.147488594055176
[0:00:06.354545] Step:580, loss:2.4012746810913086
[0:00:06.344338] Step:600, loss:1.7057498693466187
[0:00:06.330753] Step:620, loss:2.389599323272705
```

```
[0:00:06.470682] Step:40, loss:2.406553268432617
[0:00:06.368222] Step:60, loss:1.9597632884979248
[0:00:06.360373] Step:80, loss:2.01263165473938
[0:00:06.367866] Step:100, loss:2.0152368545532227
[0:00:06.433587] Step:120, loss:1.6612085103988647
[0:00:06.547826] Step:140, loss:2.1646509170532227
[0:00:06.403579] Step:160, loss:1.538785696029663
[0:00:06.533406] Step:180, loss:2.132829427719116
[0:00:06.523904] Step:200, loss:2.5021376609802246
[0:00:06.421889] Step:220, loss:3.3712172508239746
[0:00:06.413011] Step:240, loss:2.381728410720825
```