```
# -*- coding: utf-8 -*-
"""Untitled3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/
1xYp2dk_r2hk15s8jFGqgXRbpvFLQ51Y9

dataloader
"""

import threading
import random

import torch
import torch.multiprocessing as multiprocessing
from torch.utils.data import DataLoader
from torch.utils.data import SequentialSampler
from torch.utils.data import RandomSampler
from torch.utils.data import BatchSampler
from torch.utils.data import _utils
from torch.utils.data.dataloader import _DataLoaderIter

from torch.utils.data._utils import collate
from torch.utils.data._utils import signal_handling
from torch.utils.data._utils import MP_STATUS_CHECK_INTERVAL
from torch.utils.data._utils import ExceptionWrapper
from torch.utils.data._utils import IS_WINDOWS
from torch.utils.data._utils.worker import ManagerWatchdog

from torch._six import queue

def _ms_loop(dataset, index_queue, data_queue, done_event,
collate_fn, scale, seed, init_fn, worker_id):
    try:
        collate._use_shared_memory = True
        signal_handling._set_worker_signal_handlers()

        torch.set_num_threads(1)
        random.seed(seed)
        torch.manual_seed(seed)

        data_queue.cancel_join_thread()

        if init_fn is not None:
            init_fn(worker_id)

        watchdog = ManagerWatchdog()

        while watchdog.is_alive():
            try:
                r =
index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
```

```
            except queue.Empty:
                continue

            if r is None:
                assert done_event.is_set()
                return
            elif done_event.is_set():
                continue

            idx, batch_indices = r
            try:
                idx_scale = 0
                if len(scale) > 1 and dataset.train:
                    idx_scale = random.randrange(0, len(scale))
                    dataset.set_scale(idx_scale)

                samples = collate_fn([dataset[i] for i in
batch_indices])
                samples.append(idx_scale)
            except Exception:
                data_queue.put((idx,
ExceptionWrapper(sys.exc_info())))
            else:
                data_queue.put((idx, samples))
                del samples

    except KeyboardInterrupt:
        pass

class _MSDataLoaderIter(_DataLoaderIter):

    def __init__(self, loader):
        self.dataset = loader.dataset
        self.scale = loader.scale
        self.collate_fn = loader.collate_fn
        self.batch_sampler = loader.batch_sampler
        self.num_workers = loader.num_workers
        self.pin_memory = loader.pin_memory and
torch.cuda.is_available()
        self.timeout = loader.timeout

        self.sample_iter = iter(self.batch_sampler)

        base_seed = torch.LongTensor(1).random_().item()

        if self.num_workers > 0:
            self.worker_init_fn = loader.worker_init_fn
            self.worker_queue_idx = 0
            self.worker_result_queue = multiprocessing.Queue()
            self.batches_outstanding = 0
            self.worker_pids_set = False
            self.shutdown = False
            self.send_idx = 0
            self.rcvd_idx = 0
```

```
self.reorder_dict = {}
self.done_event = multiprocessing.Event()

base_seed = torch.LongTensor(1).random_()[0]

self.index_queues = []
self.workers = []
for i in range(self.num_workers):
    index_queue = multiprocessing.Queue()
    index_queue.cancel_join_thread()
    w = multiprocessing.Process(
        target=_ms_loop,
        args=(
            self.dataset,
            index_queue,
            self.worker_result_queue,
            self.done_event,
            self.collate_fn,
            self.scale,
            base_seed + i,
            self.worker_init_fn,
            i
        )
    )
    w.daemon = True
    w.start()
    self.index_queues.append(index_queue)
    self.workers.append(w)

if self.pin_memory:
    self.data_queue = queue.Queue()
    pin_memory_thread = threading.Thread(
        target=_utils.pin_memory._pin_memory_loop,
        args=(
            self.worker_result_queue,
            self.data_queue,
            torch.cuda.current_device(),
            self.done_event
        )
    )
    pin_memory_thread.daemon = True
    pin_memory_thread.start()
    self.pin_memory_thread = pin_memory_thread
else:
    self.data_queue = self.worker_result_queue

_utils.signal_handling._set_worker_pids(
    id(self), tuple(w.pid for w in self.workers)
)
_utils.signal_handling._set_SIGCHLD_handler()
self.worker_pids_set = True

for _ in range(2 * self.num_workers):
    self._put_indices()
```

```
class MSDataLoader(DataLoader):

    def __init__(self, cfg, *args, **kwargs):
        super(MSDataLoader, self).__init__(
            *args, **kwargs, num_workers=cfg.n_threads
        )
        self.scale = cfg.scale

    def __iter__(self):
        return _MSDataLoaderIter(self)

## ECCV-2018-Image Super-Resolution Using Very Deep Residual Channel
Attention Networks
## https://arxiv.org/abs/1807.02758
from model import common

import torch.nn as nn

def make_model(args, parent=False):
    return RCAN(args)

## Channel Attention (CA) Layer
class CALayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(CALayer, self).__init__()
        # global average pooling: feature --> point
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # feature channel downscale and upscale --> channel weight
        self.conv_du = nn.Sequential(
                nn.Conv2d(channel, channel // reduction, 1,
padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // reduction, channel, 1,
padding=0, bias=True),
                nn.Sigmoid()
        )

    def forward(self, x):
        y = self.avg_pool(x)
        y = self.conv_du(y)
        return x * y

## Residual Channel Attention Block (RCAB)
class RCAB(nn.Module):
    def __init__(
        self, conv, n_feat, kernel_size, reduction,
        bias=True, bn=False, act=nn.ReLU(True), res_scale=1):

        super(RCAB, self).__init__()
        modules_body = []
        for i in range(2):
            modules_body.append(conv(n_feat, n_feat, kernel_size,
```

```
bias=bias))
            if bn: modules_body.append(nn.BatchNorm2d(n_feat))
            if i == 0: modules_body.append(act)
        modules_body.append(CALayer(n_feat, reduction))
        self.body = nn.Sequential(*modules_body)
        self.res_scale = res_scale

    def forward(self, x):
        res = self.body(x)
        #res = self.body(x).mul(self.res_scale)
        res += x
        return res

## Residual Group (RG)
class ResidualGroup(nn.Module):
    def __init__(self, conv, n_feat, kernel_size, reduction, act,
res_scale, n_resblocks):
        super(ResidualGroup, self).__init__()
        modules_body = []
        modules_body = [
            RCAB(
                conv, n_feat, kernel_size, reduction, bias=True,
bn=False, act=nn.ReLU(True), res_scale=1) \
            for _ in range(n_resblocks)]
        modules_body.append(conv(n_feat, n_feat, kernel_size))
        self.body = nn.Sequential(*modules_body)

    def forward(self, x):
        res = self.body(x)
        res += x
        return res

## Residual Channel Attention Network (RCAN)
class RCAN(nn.Module):
    def __init__(self, args, conv=common.default_conv):
        super(RCAN, self).__init__()

        n_resgroups = args.n_resgroups
        n_resblocks = args.n_resblocks
        n_feats = args.n_feats
        kernel_size = 3
        reduction = args.reduction
        scale = args.scale[0]
        act = nn.ReLU(True)

        # RGB mean for DIV2K
        self.sub_mean = common.MeanShift(args.rgb_range)

        # define head module
        modules_head = [conv(args.n_colors, n_feats, kernel_size)]

        # define body module
        modules_body = [
            ResidualGroup(
```

```
                conv, n_feats, kernel_size, reduction, act=act,
res_scale=args.res_scale, n_resblocks=n_resblocks) \
            for _ in range(n_resgroups)]

        modules_body.append(conv(n_feats, n_feats, kernel_size))

        # define tail module
        modules_tail = [
            common.Upsampler(conv, scale, n_feats, act=False),
            conv(n_feats, args.n_colors, kernel_size)]

        self.add_mean = common.MeanShift(args.rgb_range, sign=1)

        self.head = nn.Sequential(*modules_head)
        self.body = nn.Sequential(*modules_body)
        self.tail = nn.Sequential(*modules_tail)

    def forward(self, x):
        x = self.sub_mean(x)
        x = self.head(x)

        res = self.body(x)
        res += x

        x = self.tail(res)
        x = self.add_mean(x)

        return x

    def load_state_dict(self, state_dict, strict=False):
        own_state = self.state_dict()
        for name, param in state_dict.items():
            if name in own_state:
                if isinstance(param, nn.Parameter):
                    param = param.data
                try:
                    own_state[name].copy_(param)
                except Exception:
                    if name.find('tail') >= 0:
                        print('Replace pre-trained upsampler to new
one...')
                    else:
                        raise RuntimeError('While copying the
parameter named {}, '
                                           'whose dimensions in the
model are {} and '
                                           'whose dimensions in the
checkpoint are {}.'
                                           .format(name,
own_state[name].size(), param.size()))
            elif strict:
                if name.find('tail') == -1:
                    raise KeyError('unexpected key "{}" in
state_dict'
```

```
                                    .format(name))

        if strict:
            missing = set(own_state.keys()) - set(state_dict.keys())
            if len(missing) > 0:
                raise KeyError('missing keys in state_dict:
"{}"'.format(missing))

"""optinal"""

import argparse
import template

parser = argparse.ArgumentParser(description='EDSR and MDSR')

parser.add_argument('--debug', action='store_true',
                    help='Enables debug mode')
parser.add_argument('--template', default='.',
                    help='You can set various templates in
option.py')

# Hardware specifications
parser.add_argument('--n_threads', type=int, default=6,
                    help='number of threads for data loading')
parser.add_argument('--cpu', action='store_true',
                    help='use cpu only')
parser.add_argument('--n_GPUs', type=int, default=1,
                    help='number of GPUs')
parser.add_argument('--seed', type=int, default=1,
                    help='random seed')

# Data specifications
parser.add_argument('--dir_data', type=str, default='/Users/ziyang/
Desktop/541data_set',
                    help='dataset directory')
parser.add_argument('--dir_demo', type=str, default='/Users/ziyang/
Desktop/541data_set/test',
                    help='demo image directory')
parser.add_argument('--data_train', type=str, default='DIV2K',
                    help='train dataset name')
parser.add_argument('--data_test', type=str, default='DIV2K',
                    help='test dataset name')
parser.add_argument('--data_range', type=str,
default='1-800/801-810',
                    help='train/test data range')
parser.add_argument('--ext', type=str, default='sep',
                    help='dataset file extension')
parser.add_argument('--scale', type=str, default='4',
                    help='super resolution scale')
parser.add_argument('--patch_size', type=int, default=192,
                    help='output patch size')
parser.add_argument('--rgb_range', type=int, default=255,
                    help='maximum value of RGB')
parser.add_argument('--n_colors', type=int, default=3,
```

```
                    help='number of color channels to use')
parser.add_argument('--chop', action='store_true',
                    help='enable memory-efficient forward')
parser.add_argument('--no_augment', action='store_true',
                    help='do not use data augmentation')

# Model specifications
parser.add_argument('--model', default='EDSR',
                    help='model name')

parser.add_argument('--act', type=str, default='relu',
                    help='activation function')
parser.add_argument('--pre_train', type=str, default='/Users/ziyang/
Desktop/EDSR-PyTorch-master/experiment/test/model/',
                    help='pre-trained model directory')
parser.add_argument('--extend', type=str, default='/Users/ziyang/
Desktop/EDSR-PyTorch-master/experiment/test/model/',
                    help='pre-trained model directory')
parser.add_argument('--n_resblocks', type=int, default=16,
                    help='number of residual blocks')
parser.add_argument('--n_feats', type=int, default=64,
                    help='number of feature maps')
parser.add_argument('--res_scale', type=float, default=1,
                    help='residual scaling')
parser.add_argument('--shift_mean', default=True,
                    help='subtract pixel mean from the input')
parser.add_argument('--dilation', action='store_true',
                    help='use dilated convolution')
parser.add_argument('--precision', type=str, default='single',
                    choices=('single', 'half'),
                    help='FP precision for test (single | half)')

# Option for Residual dense network (RDN)
parser.add_argument('--G0', type=int, default=64,
                    help='default number of filters. (Use in RDN)')
parser.add_argument('--RDNkSize', type=int, default=3,
                    help='default kernel size. (Use in RDN)')
parser.add_argument('--RDNconfig', type=str, default='B',
                    help='parameters config of RDN. (Use in RDN)')

# Option for Residual channel attention network (RCAN)
parser.add_argument('--n_resgroups', type=int, default=10,
                    help='number of residual groups')
parser.add_argument('--reduction', type=int, default=16,
                    help='number of feature maps reduction')

# Training specifications
parser.add_argument('--reset', action='store_true',
                    help='reset the training')
parser.add_argument('--test_every', type=int, default=1000,
                    help='do test per every N batches')
parser.add_argument('--epochs', type=int, default=300,
                    help='number of epochs to train')
parser.add_argument('--batch_size', type=int, default=16,
```

```
                    help='input batch size for training')
parser.add_argument('--split_batch', type=int, default=1,
                    help='split the batch into smaller chunks')
parser.add_argument('--self_ensemble', action='store_true',
                    help='use self-ensemble method for test')
parser.add_argument('--test_only', action='store_true',
                    help='set this option to test the model')
parser.add_argument('--gan_k', type=int, default=1,
                    help='k value for adversarial loss')

# Optimization specifications
parser.add_argument('--lr', type=float, default=1e-4,
                    help='learning rate')
parser.add_argument('--decay', type=str, default='200',
                    help='learning rate decay type')
parser.add_argument('--gamma', type=float, default=0.5,
                    help='learning rate decay factor for step
decay')
parser.add_argument('--optimizer', default='ADAM',
                    choices=('SGD', 'ADAM', 'RMSprop'),
                    help='optimizer to use (SGD | ADAM | RMSprop)')
parser.add_argument('--momentum', type=float, default=0.9,
                    help='SGD momentum')
parser.add_argument('--betas', type=tuple, default=(0.9, 0.999),
                    help='ADAM beta')
parser.add_argument('--epsilon', type=float, default=1e-8,
                    help='ADAM epsilon for numerical stability')
parser.add_argument('--weight_decay', type=float, default=0,
                    help='weight decay')
parser.add_argument('--gclip', type=float, default=0,
                    help='gradient clipping threshold (0 = no
clipping)')

# Loss specifications
parser.add_argument('--loss', type=str, default='1*L1',
                    help='loss function configuration')
parser.add_argument('--skip_threshold', type=float, default='1e8',
                    help='skipping batch that has large error')

# Log specifications
parser.add_argument('--save', type=str, default='test',
                    help='file name to save')
parser.add_argument('--load', type=str, default='',
                    help='file name to load')
parser.add_argument('--resume', type=int, default=0,
                    help='resume from specific checkpoint')
parser.add_argument('--save_models', action='store_true',
                    help='save all intermediate models')
parser.add_argument('--print_every', type=int, default=100,
                    help='how many batches to wait before logging
training status')
parser.add_argument('--save_results', action='store_true',
                    help='save output results')
parser.add_argument('--save_gt', action='store_true',
```

```
                    help='save low-resolution and high-resolution
images together')

args = parser.parse_args()
template.set_template(args)

args.scale = list(map(lambda x: int(x), args.scale.split('+')))
args.data_train = args.data_train.split('+')
args.data_test = args.data_test.split('+')

if args.epochs == 0:
    args.epochs = 1e8

for arg in vars(args):
    if vars(args)[arg] == 'True':
        vars(args)[arg] = True
    elif vars(args)[arg] == 'False':
        vars(args)[arg] = False

"""rdn"""

# Residual Dense Network for Image Super-Resolution
# https://arxiv.org/abs/1802.08797

from model import common

import torch
import torch.nn as nn


def make_model(args, parent=False):
    return RDN(args)

class RDB_Conv(nn.Module):
    def __init__(self, inChannels, growRate, kSize=3):
        super(RDB_Conv, self).__init__()
        Cin = inChannels
        G  = growRate
        self.conv = nn.Sequential(*[
            nn.Conv2d(Cin, G, kSize, padding=(kSize-1)//2,
stride=1),
            nn.ReLU()
        ])

    def forward(self, x):
        out = self.conv(x)
        return torch.cat((x, out), 1)

class RDB(nn.Module):
    def __init__(self, growRate0, growRate, nConvLayers, kSize=3):
        super(RDB, self).__init__()
        G0 = growRate0
        G  = growRate
        C  = nConvLayers
```

```
        convs = []
        for c in range(C):
            convs.append(RDB_Conv(G0 + c*G, G))
        self.convs = nn.Sequential(*convs)

        # Local Feature Fusion
        self.LFF = nn.Conv2d(G0 + C*G, G0, 1, padding=0, stride=1)

    def forward(self, x):
        return self.LFF(self.convs(x)) + x

class RDN(nn.Module):
    def __init__(self, args):
        super(RDN, self).__init__()
        r = args.scale[0]
        G0 = args.G0
        kSize = args.RDNkSize

        # number of RDB blocks, conv layers, out channels
        self.D, C, G = {
            'A': (20, 6, 32),
            'B': (16, 8, 64),
        }[args.RDNconfig]

        # Shallow feature extraction net
        self.SFENet1 = nn.Conv2d(args.n_colors, G0, kSize,
padding=(kSize-1)//2, stride=1)
        self.SFENet2 = nn.Conv2d(G0, G0, kSize, padding=(kSize-1)//
2, stride=1)

        # Redidual dense blocks and dense feature fusion
        self.RDBs = nn.ModuleList()
        for i in range(self.D):
            self.RDBs.append(
                RDB(growRate0 = G0, growRate = G, nConvLayers = C)
            )

        # Global Feature Fusion
        self.GFF = nn.Sequential(*[
            nn.Conv2d(self.D * G0, G0, 1, padding=0, stride=1),
            nn.Conv2d(G0, G0, kSize, padding=(kSize-1)//2, stride=1)
        ])

        # Up-sampling net
        if r == 2 or r == 3:
            self.UPNet = nn.Sequential(*[
                nn.Conv2d(G0, G * r * r, kSize, padding=(kSize-1)//
2, stride=1),
                nn.PixelShuffle(r),
                nn.Conv2d(G, args.n_colors, kSize,
padding=(kSize-1)//2, stride=1)
            ])
        elif r == 4:
```

```
            self.UPNet = nn.Sequential(*[
                nn.Conv2d(G0, G * 4, kSize, padding=(kSize-1)//2,
stride=1),
                nn.PixelShuffle(2),
                nn.Conv2d(G, G * 4, kSize, padding=(kSize-1)//2,
stride=1),
                nn.PixelShuffle(2),
                nn.Conv2d(G, args.n_colors, kSize,
padding=(kSize-1)//2, stride=1)
            ])
        else:
            raise ValueError("scale must be 2 or 3 or 4.")

    def forward(self, x):
        f__1 = self.SFENet1(x)
        x  = self.SFENet2(f__1)

        RDBs_out = []
        for i in range(self.D):
            x = self.RDBs[i](x)
            RDBs_out.append(x)

        x = self.GFF(torch.cat(RDBs_out,1))
        x += f__1

        return self.UPNet(x)

"""mdsr"""

from model import common

import torch.nn as nn

url = {
    'r16f64': 'https://cv.snu.ac.kr/research/EDSR/models/
mdsr_baseline-a00cab12.pt',
    'r80f64': 'https://cv.snu.ac.kr/research/EDSR/models/
mdsr-4a78bedf.pt'
}

def make_model(args, parent=False):
    return MDSR(args)

class MDSR(nn.Module):
    def __init__(self, args, conv=common.default_conv):
        super(MDSR, self).__init__()
        n_resblocks = args.n_resblocks
        n_feats = args.n_feats
        kernel_size = 3
        act = nn.ReLU(True)
        self.scale_idx = 0
        self.url = url['r{}f{}'.format(n_resblocks, n_feats)]
        self.sub_mean = common.MeanShift(args.rgb_range)
        self.add_mean = common.MeanShift(args.rgb_range, sign=1)
```

```
        m_head = [conv(args.n_colors, n_feats, kernel_size)]

        self.pre_process = nn.ModuleList([
            nn.Sequential(
                common.ResBlock(conv, n_feats, 5, act=act),
                common.ResBlock(conv, n_feats, 5, act=act)
            ) for _ in args.scale
        ])

        m_body = [
            common.ResBlock(
                conv, n_feats, kernel_size, act=act
            ) for _ in range(n_resblocks)
        ]
        m_body.append(conv(n_feats, n_feats, kernel_size))

        self.upsample = nn.ModuleList([
            common.Upsampler(conv, s, n_feats, act=False) for s in
args.scale
        ])

        m_tail = [conv(n_feats, args.n_colors, kernel_size)]

        self.head = nn.Sequential(*m_head)
        self.body = nn.Sequential(*m_body)
        self.tail = nn.Sequential(*m_tail)

    def forward(self, x):
        x = self.sub_mean(x)
        x = self.head(x)
        x = self.pre_process[self.scale_idx](x)

        res = self.body(x)
        res += x

        x = self.upsample[self.scale_idx](res)
        x = self.tail(x)
        x = self.add_mean(x)

        return x

    def set_scale(self, scale_idx):
        self.scale_idx = scale_idx

"""ddbpn"""

# Deep Back-Projection Networks For Super-Resolution
# https://arxiv.org/abs/1803.02735

from model import common

import torch
import torch.nn as nn
```

```
def make_model(args, parent=False):
    return DDBPN(args)

def projection_conv(in_channels, out_channels, scale, up=True):
    kernel_size, stride, padding = {
        2: (6, 2, 2),
        4: (8, 4, 2),
        8: (12, 8, 2)
    }[scale]
    if up:
        conv_f = nn.ConvTranspose2d
    else:
        conv_f = nn.Conv2d

    return conv_f(
        in_channels, out_channels, kernel_size,
        stride=stride, padding=padding
    )

class DenseProjection(nn.Module):
    def __init__(self, in_channels, nr, scale, up=True,
bottleneck=True):
        super(DenseProjection, self).__init__()
        if bottleneck:
            self.bottleneck = nn.Sequential(*[
                nn.Conv2d(in_channels, nr, 1),
                nn.PReLU(nr)
            ])
            inter_channels = nr
        else:
            self.bottleneck = None
            inter_channels = in_channels

        self.conv_1 = nn.Sequential(*[
            projection_conv(inter_channels, nr, scale, up),
            nn.PReLU(nr)
        ])
        self.conv_2 = nn.Sequential(*[
            projection_conv(nr, inter_channels, scale, not up),
            nn.PReLU(inter_channels)
        ])
        self.conv_3 = nn.Sequential(*[
            projection_conv(inter_channels, nr, scale, up),
            nn.PReLU(nr)
        ])

    def forward(self, x):
        if self.bottleneck is not None:
            x = self.bottleneck(x)

        a_0 = self.conv_1(x)
        b_0 = self.conv_2(a_0)
```

```
        e = b_0.sub(x)
        a_1 = self.conv_3(e)

        out = a_0.add(a_1)

        return out

class DDBPN(nn.Module):
    def __init__(self, args):
        super(DDBPN, self).__init__()
        scale = args.scale[0]

        n0 = 128
        nr = 32
        self.depth = 6

        rgb_mean = (0.4488, 0.4371, 0.4040)
        rgb_std = (1.0, 1.0, 1.0)
        self.sub_mean = common.MeanShift(args.rgb_range, rgb_mean,
rgb_std)
        initial = [
            nn.Conv2d(args.n_colors, n0, 3, padding=1),
            nn.PReLU(n0),
            nn.Conv2d(n0, nr, 1),
            nn.PReLU(nr)
        ]
        self.initial = nn.Sequential(*initial)

        self.upmodules = nn.ModuleList()
        self.downmodules = nn.ModuleList()
        channels = nr
        for i in range(self.depth):
            self.upmodules.append(
                DenseProjection(channels, nr, scale, True, i > 1)
            )
            if i != 0:
                channels += nr

        channels = nr
        for i in range(self.depth - 1):
            self.downmodules.append(
                DenseProjection(channels, nr, scale, False, i != 0)
            )
            channels += nr

        reconstruction = [
            nn.Conv2d(self.depth * nr, args.n_colors, 3, padding=1)
        ]
        self.reconstruction = nn.Sequential(*reconstruction)

        self.add_mean = common.MeanShift(args.rgb_range, rgb_mean,
rgb_std, 1)

    def forward(self, x):
```

```
        x = self.sub_mean(x)
        x = self.initial(x)

        h_list = []
        l_list = []
        for i in range(self.depth - 1):
            if i == 0:
                l = x
            else:
                l = torch.cat(l_list, dim=1)
            h_list.append(self.upmodules[i](l))
            l_list.append(self.downmodules[i](torch.cat(h_list,
dim=1)))

        h_list.append(self.upmodules[-1](torch.cat(l_list, dim=1)))
        out = self.reconstruction(torch.cat(h_list, dim=1))
        out = self.add_mean(out)

        return out

"""model"""

import math

import torch
import torch.nn as nn
import torch.nn.functional as F

def default_conv(in_channels, out_channels, kernel_size, bias=True):
    return nn.Conv2d(
        in_channels, out_channels, kernel_size,
        padding=(kernel_size//2), bias=bias)

class MeanShift(nn.Conv2d):
    def __init__(
        self, rgb_range,
        rgb_mean=(0.4488, 0.4371, 0.4040), rgb_std=(1.0, 1.0, 1.0),
sign=-1):

        super(MeanShift, self).__init__(3, 3, kernel_size=1)
        std = torch.Tensor(rgb_std)
        self.weight.data = torch.eye(3).view(3, 3, 1, 1) /
std.view(3, 1, 1, 1)
        self.bias.data = sign * rgb_range * torch.Tensor(rgb_mean) /
std
        for p in self.parameters():
            p.requires_grad = False

class BasicBlock(nn.Sequential):
    def __init__(
        self, conv, in_channels, out_channels, kernel_size,
stride=1, bias=False,
        bn=True, act=nn.ReLU(True)):
```

```
        m = [conv(in_channels, out_channels, kernel_size,
bias=bias)]
        if bn:
            m.append(nn.BatchNorm2d(out_channels))
        if act is not None:
            m.append(act)

        super(BasicBlock, self).__init__(*m)

class ResBlock(nn.Module):
    def __init__(
        self, conv, n_feats, kernel_size,
        bias=True, bn=False, act=nn.ReLU(True), res_scale=1):

        super(ResBlock, self).__init__()
        m = []
        for i in range(2):
            m.append(conv(n_feats, n_feats, kernel_size, bias=bias))
            if bn:
                m.append(nn.BatchNorm2d(n_feats))
            if i == 0:
                m.append(act)

        self.body = nn.Sequential(*m)
        self.res_scale = res_scale

    def forward(self, x):
        res = self.body(x).mul(self.res_scale)
        res += x

        return res

class Upsampler(nn.Sequential):
    def __init__(self, conv, scale, n_feats, bn=False, act=False,
bias=True):

        m = []
        if (scale & (scale - 1)) == 0:    # Is scale = 2^n?
            for _ in range(int(math.log(scale, 2))):
                m.append(conv(n_feats, 4 * n_feats, 3, bias))
                m.append(nn.PixelShuffle(2))
                if bn:
                    m.append(nn.BatchNorm2d(n_feats))
                if act == 'relu':
                    m.append(nn.ReLU(True))
                elif act == 'prelu':
                    m.append(nn.PReLU(n_feats))

        elif scale == 3:
            m.append(conv(n_feats, 9 * n_feats, 3, bias))
            m.append(nn.PixelShuffle(3))
            if bn:
                m.append(nn.BatchNorm2d(n_feats))
            if act == 'relu':
```

```
                m.append(nn.ReLU(True))
            elif act == 'prelu':
                m.append(nn.PReLU(n_feats))
        else:
            raise NotImplementedError

        super(Upsampler, self).__init__(*m)

import os
from importlib import import_module

import torch
import torch.nn as nn
import torch.nn.parallel as P
import torch.utils.model_zoo

class Model(nn.Module):
    def __init__(self, args, ckp):
        super(Model, self).__init__()
        print('Making model...')

        self.scale = args.scale
        self.idx_scale = 0
        self.input_large = (args.model == 'VDSR')
        self.self_ensemble = args.self_ensemble
        self.chop = args.chop
        self.precision = args.precision
        self.cpu = args.cpu
        self.device = torch.device('cpu' if args.cpu else 'cuda')
        self.n_GPUs = args.n_GPUs
        self.save_models = args.save_models

        module = import_module('model.' + args.model.lower())
        self.model = module.make_model(args).to(self.device)
        if args.precision == 'half':
            self.model.half()

        self.load(
            ckp.get_path('model'),
            pre_train=args.pre_train,
            resume=args.resume,
            cpu=args.cpu
        )
        print(self.model, file=ckp.log_file)

    def forward(self, x, idx_scale):
        self.idx_scale = idx_scale
        if hasattr(self.model, 'set_scale'):
            self.model.set_scale(idx_scale)

        if self.training:
            if self.n_GPUs > 1:
                return P.data_parallel(self.model, x,
range(self.n_GPUs))
```

```
            else:
                return self.model(x)
        else:
            if self.chop:
                forward_function = self.forward_chop
            else:
                forward_function = self.model.forward

            if self.self_ensemble:
                return self.forward_x8(x,
forward_function=forward_function)
            else:
                return forward_function(x)

    def save(self, apath, epoch, is_best=False):
        save_dirs = [os.path.join(apath, 'model_latest.pt')]

        if is_best:
            save_dirs.append(os.path.join(apath, 'model_best.pt'))
        if self.save_models:
            save_dirs.append(
                os.path.join(apath, 'model_{}.pt'.format(epoch))
            )

        for s in save_dirs:
            torch.save(self.model.state_dict(), s)

    def load(self, apath, pre_train='', resume=-1, cpu=False):
        load_from = None
        kwargs = {}
        if cpu:
            kwargs = {'map_location': lambda storage, loc: storage}

        if resume == -1:
            load_from = torch.load(
                os.path.join(apath, 'model_latest.pt'),
                **kwargs
            )
        elif resume == 0:
            if pre_train == 'download':
                print('Download the model')
                dir_model = os.path.join('..', 'models')
                os.makedirs(dir_model, exist_ok=True)
                load_from = torch.utils.model_zoo.load_url(
                    self.model.url,
                    model_dir=dir_model,
                    **kwargs
                )
            elif pre_train:
                print('Load the model from {}'.format(pre_train))
                load_from = torch.load(pre_train, **kwargs)
        else:
            load_from = torch.load(
                os.path.join(apath, 'model_{}.pt'.format(resume)),
```

```
                **kwargs
            )

        if load_from:
            self.model.load_state_dict(load_from, strict=False)

    def forward_chop(self, *args, shave=10, min_size=160000):
        scale = 1 if self.input_large else
self.scale[self.idx_scale]
        n_GPUs = min(self.n_GPUs, 4)
        # height, width
        h, w = args[0].size()[-2:]

        top = slice(0, h//2 + shave)
        bottom = slice(h - h//2 - shave, h)
        left = slice(0, w//2 + shave)
        right = slice(w - w//2 - shave, w)
        x_chops = [torch.cat([
            a[..., top, left],
            a[..., top, right],
            a[..., bottom, left],
            a[..., bottom, right]
        ]) for a in args]

        y_chops = []
        if h * w < 4 * min_size:
            for i in range(0, 4, n_GPUs):
                x = [x_chop[i:(i + n_GPUs)] for x_chop in x_chops]
                y = P.data_parallel(self.model, *x, range(n_GPUs))
                if not isinstance(y, list): y = [y]
                if not y_chops:
                    y_chops = [[c for c in _y.chunk(n_GPUs, dim=0)]
for _y in y]
                else:
                    for y_chop, _y in zip(y_chops, y):
                        y_chop.extend(_y.chunk(n_GPUs, dim=0))
        else:
            for p in zip(*x_chops):
                y = self.forward_chop(*p, shave=shave,
min_size=min_size)
                if not isinstance(y, list): y = [y]
                if not y_chops:
                    y_chops = [[_y] for _y in y]
                else:
                    for y_chop, _y in zip(y_chops, y):
y_chop.append(_y)

        h *= scale
        w *= scale
        top = slice(0, h//2)
        bottom = slice(h - h//2, h)
        bottom_r = slice(h//2 - h, None)
        left = slice(0, w//2)
        right = slice(w - w//2, w)
```

```
        right_r = slice(w//2 - w, None)

        # batch size, number of color channels
        b, c = y_chops[0][0].size()[:-2]
        y = [y_chop[0].new(b, c, h, w) for y_chop in y_chops]
        for y_chop, _y in zip(y_chops, y):
            _y[..., top, left] = y_chop[0][..., top, left]
            _y[..., top, right] = y_chop[1][..., top, right_r]
            _y[..., bottom, left] = y_chop[2][..., bottom_r, left]
            _y[..., bottom, right] = y_chop[3][..., bottom_r,
right_r]

        if len(y) == 1: y = y[0]

        return y

    def forward_x8(self, *args, forward_function=None):
        def _transform(v, op):
            if self.precision != 'single': v = v.float()

            v2np = v.data.cpu().numpy()
            if op == 'v':
                tfnp = v2np[:, :, :, ::-1].copy()
            elif op == 'h':
                tfnp = v2np[:, :, ::-1, :].copy()
            elif op == 't':
                tfnp = v2np.transpose((0, 1, 3, 2)).copy()

            ret = torch.Tensor(tfnp).to(self.device)
            if self.precision == 'half': ret = ret.half()

            return ret

        list_x = []
        for a in args:
            x = [a]
            for tf in 'v', 'h', 't': x.extend([_transform(_x, tf)
for _x in x])

            list_x.append(x)

        list_y = []
        for x in zip(*list_x):
            y = forward_function(*x)
            if not isinstance(y, list): y = [y]
            if not list_y:
                list_y = [[_y] for _y in y]
            else:
                for _list_y, _y in zip(list_y, y):
_list_y.append(_y)

        for _list_y in list_y:
            for i in range(len(_list_y)):
                if i > 3:
```

```
                    _list_y[i] = _transform(_list_y[i], 't')
                if i % 4 > 1:
                    _list_y[i] = _transform(_list_y[i], 'h')
                if (i % 4) % 2 == 1:
                    _list_y[i] = _transform(_list_y[i], 'v')

        y = [torch.cat(_y, dim=0).mean(dim=0, keepdim=True) for _y
in list_y]
        if len(y) == 1: y = y[0]

        return y

"""trainer"""

import os
import math
from decimal import Decimal

import utility

import torch
import torch.nn.utils as utils
from tqdm import tqdm

class Trainer():
    def __init__(self, args, loader, my_model, my_loss, ckp):
        self.args = args
        self.scale = args.scale

        self.ckp = ckp
        self.loader_train = loader.loader_train
        self.loader_test = loader.loader_test
        self.model = my_model
        self.loss = my_loss
        self.optimizer = utility.make_optimizer(args, self.model)

        if self.args.load != '':
            self.optimizer.load(ckp.dir, epoch=len(ckp.log))

        self.error_last = 1e8

    def train(self):
        self.loss.step()
        epoch = self.optimizer.get_last_epoch() + 1
        lr = self.optimizer.get_lr()

        self.ckp.write_log(
            '[Epoch {}]\tLearning rate: {:.2e}'.format(epoch,
Decimal(lr))
        )
        self.loss.start_log()
        self.model.train()

        timer_data, timer_model = utility.timer(), utility.timer()
```

```
        # TEMP
        self.loader_train.dataset.set_scale(0)
        for batch, (lr, hr, _,) in enumerate(self.loader_train):
            lr, hr = self.prepare(lr, hr)
            timer_data.hold()
            timer_model.tic()

            self.optimizer.zero_grad()
            sr = self.model(lr, 0)
            loss = self.loss(sr, hr)
            loss.backward()
            if self.args.gclip > 0:
                utils.clip_grad_value_(
                    self.model.parameters(),
                    self.args.gclip
                )
            self.optimizer.step()

            timer_model.hold()

            if (batch + 1) % self.args.print_every == 0:
                self.ckp.write_log('[{}/{}]\t{}\t{:.1f}+{:.1f}
s'.format(
                    (batch + 1) * self.args.batch_size,
                    len(self.loader_train.dataset),
                    self.loss.display_loss(batch),
                    timer_model.release(),
                    timer_data.release()))

            timer_data.tic()

        self.loss.end_log(len(self.loader_train))
        self.error_last = self.loss.log[-1, -1]
        self.optimizer.schedule()

    def test(self):
        torch.set_grad_enabled(False)

        epoch = self.optimizer.get_last_epoch()
        self.ckp.write_log('\nEvaluation:')
        self.ckp.add_log(
            torch.zeros(1, len(self.loader_test), len(self.scale))
        )
        self.model.eval()

        timer_test = utility.timer()
        if self.args.save_results: self.ckp.begin_background()
        for idx_data, d in enumerate(self.loader_test):
            for idx_scale, scale in enumerate(self.scale):
                d.dataset.set_scale(idx_scale)
                for lr, hr, filename in tqdm(d, ncols=80):
                    lr, hr = self.prepare(lr, hr)
                    sr = self.model(lr, idx_scale)
                    sr = utility.quantize(sr, self.args.rgb_range)
```

```
                    save_list = [sr]
                    self.ckp.log[-1, idx_data, idx_scale] +=
utility.calc_psnr(
                        sr, hr, scale, self.args.rgb_range,
dataset=d
                    )
                    if self.args.save_gt:
                        save_list.extend([lr, hr])

                    if self.args.save_results:
                        self.ckp.save_results(d, filename[0],
save_list, scale)

                self.ckp.log[-1, idx_data, idx_scale] /= len(d)
                best = self.ckp.log.max(0)
                self.ckp.write_log(
                    '[{} x{}]\tPSNR: {:.3f} (Best: {:.3f} @epoch
{})'.format(
                        d.dataset.name,
                        scale,
                        self.ckp.log[-1, idx_data, idx_scale],
                        best[0][idx_data, idx_scale],
                        best[1][idx_data, idx_scale] + 1
                    )
                )

        self.ckp.write_log('Forward: {:.2f}
s\n'.format(timer_test.toc()))
        self.ckp.write_log('Saving...')

        if self.args.save_results:
            self.ckp.end_background()

        if not self.args.test_only:
            self.ckp.save(self, epoch, is_best=(best[1][0, 0] + 1 ==
epoch))

        self.ckp.write_log(
            'Total: {:.2f}s\n'.format(timer_test.toc()),
refresh=True
        )

        torch.set_grad_enabled(True)

    def prepare(self, *args):
        device = torch.device('cpu' if self.args.cpu else 'cuda')
        def _prepare(tensor):
            if self.args.precision == 'half': tensor = tensor.half()
            return tensor.to(device)

        return [_prepare(a) for a in args]

    def terminate(self):
```

```
        if self.args.test_only:
            self.test()
            return True
        else:
            epoch = self.optimizer.get_last_epoch() + 1
            return epoch >= self.args.epochs

"""template"""

def set_template(args):
    # Set the templates here
    if args.template.find('jpeg') >= 0:
        args.data_train = 'DIV2K_jpeg'
        args.data_test = 'DIV2K_jpeg'
        args.epochs = 200
        args.decay = '100'

    if args.template.find('EDSR_paper') >= 0:
        args.model = 'EDSR'
        args.n_resblocks = 32
        args.n_feats = 256
        args.res_scale = 0.1

    if args.template.find('MDSR') >= 0:
        args.model = 'MDSR'
        args.patch_size = 48
        args.epochs = 650

    if args.template.find('DDBPN') >= 0:
        args.model = 'DDBPN'
        args.patch_size = 128
        args.scale = '4'

        args.data_test = 'Set5'

        args.batch_size = 20
        args.epochs = 1000
        args.decay = '500'
        args.gamma = 0.1
        args.weight_decay = 1e-4

        args.loss = '1*MSE'

    if args.template.find('GAN') >= 0:
        args.epochs = 200
        args.lr = 5e-5
        args.decay = '150'

    if args.template.find('RCAN') >= 0:
        args.model = 'RCAN'
        args.n_resgroups = 10
        args.n_resblocks = 20
        args.n_feats = 64
        args.chop = True
```

```
if args.template.find('VDSR') >= 0:
    args.model = 'VDSR'
    args.n_resblocks = 20
    args.n_feats = 64
    args.patch_size = 41
    args.lr = 1e-1
```

Importing required libraries

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import applications, Model, losses, layers, optimizers
import tensorflow_datasets as tfds

import os

import numpy as np
import matplotlib.pyplot as plt
```

Config values

In [ ]:
```
# [========= Data Preprocessing =========]
HR_SIZE = 128
SCALE = 4
LR_SIZE = int(HR_SIZE / 4)
BATCH_SIZE = 8

GEN_FILTERS = 64
DISC_FILTERS = 64
```

Data preprocessing and augmentation

In [ ]:
```
# [=====================================================]
# [================ Random Compressions ===============]
# [=====================================================]

def random_compression(example):
    hr = example['hr']
    hr_shape = tf.shape(hr)
    compression_idx = tf.random.uniform(shape = (), maxval = 7, dtype = tf.int32)

    if compression_idx == 0 or compression_idx == 1:
        # bicubic
        lr = tf.image.resize(hr, [int(hr_shape[0] / SCALE), int(hr_shape[1] / SCALE)], method = 'bicubic')
        lr = tf.cast(tf.round(tf.clip_by_value(lr, 0, 255)), tf.uint8)
    elif compression_idx == 2 or compression_idx == 3:
        # bilinear
        lr = tf.image.resize(hr, [int(hr_shape[0] / SCALE), int(hr_shape[1] / SCALE)], method = 'bilinear')
        lr = tf.cast(tf.round(tf.clip_by_value(lr, 0, 255)), tf.uint8)
    elif compression_idx == 4 or compression_idx == 5:
        # nearest
        lr = tf.image.resize(hr, [int(hr_shape[0] / SCALE), int(hr_shape[1] / SCALE)], method = 'nearest')
        lr = tf.cast(tf.round(tf.clip_by_value(lr, 0, 255)), tf.uint8)
    else:
        # default
        lr = example['lr']

    return lr, hr

# [======================================================]
# [============= Spatial Random Augmentations ===========]
# [======================================================]

@tf.function()
def random_crop(lr, hr):
    lr_shape = tf.shape(lr)[:2]

    lr_w = tf.random.uniform(shape = (), maxval = lr_shape[1] - LR_SIZE + 1, dtype = tf.int32)
    lr_h = tf.random.uniform(shape = (), maxval = lr_shape[0] - LR_SIZE + 1, dtype = tf.int32)

    hr_w = lr_w * int(SCALE)
    hr_h = lr_h * int(SCALE)

    lr_cropped = lr[lr_h:lr_h + LR_SIZE, lr_w: lr_w + LR_SIZE]
    hr_cropped = hr[hr_h:hr_h + HR_SIZE, hr_w: hr_w + HR_SIZE]

    return lr_cropped, hr_cropped

@tf.function()
def random_rotate(lr, hr):
    rn = tf.random.uniform(shape = (), maxval = 4, dtype = tf.int32)
    return tf.image.rot90(lr, rn), tf.image.rot90(hr, rn)

@tf.function()
def random_spatial_augmentation(lrs, hrs):
    lrs, hrs = tf.cond(
        tf.random.uniform(shape = (), maxval = 1) < 0.5,
        lambda: (lrs, hrs),
        lambda: random_rotate(lrs, hrs)
    )

    return tf.cast(lrs, tf.float32), tf.cast(hrs, tf.float32)
```

Downloading dataset and creating data loader

In [ ]:
```
train_data = tfds.load(f'div2k/bicubic_x{SCALE}', split = 'train', shuffle_files = True)
train_data = train_data.map(random_compression, num_parallel_calls = tf.data.AUTOTUNE)
train_data = train_data.map(random_crop, num_parallel_calls = tf.data.AUTOTUNE)
train_data = train_data.batch(BATCH_SIZE, drop_remainder = True)
train_data = train_data.map(random_spatial_augmentation, num_parallel_calls = tf.data.AUTOTUNE)

train_data = train_data.prefetch(tf.data.AUTOTUNE)
```

```
Downloading and preparing dataset div2k/bicubic_x4/2.0.0 (download: 3.97 GiB, generated: Unknown size, total:
 3.97 GiB) to /root/tensorflow_datasets/div2k/bicubic_x4/2.0.0...
EXTRACTING {'train_lr_url': 'https://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_LR_bicubic_X4.zip', 'valid_lr
_url': 'https://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_valid_LR_bicubic_X4.zip', 'train_hr_url': 'https://data.
vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_HR.zip', 'valid_hr_url': 'https://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2
K_valid_HR.zip'}

Shuffling and writing examples to /root/tensorflow_datasets/div2k/bicubic_x4/2.0.0.incompleteFFC2SO/div2k-trai
n.tfrecord

Shuffling and writing examples to /root/tensorflow_datasets/div2k/bicubic_x4/2.0.0.incompleteFFC2SO/div2k-valid
ation.tfrecord

Dataset div2k downloaded and prepared to /root/tensorflow_datasets/div2k/bicubic_x4/2.0.0. Subsequent calls wil
l reuse this data.
```

In [ ]:
```
for lrs, hrs in train_data:
    break

print(lrs.shape, hrs.shape)
print(lrs.dtype, hrs.dtype)
print(tf.reduce_min(lrs), tf.reduce_max(lrs))
print(tf.reduce_min(hrs), tf.reduce_max(hrs))
```

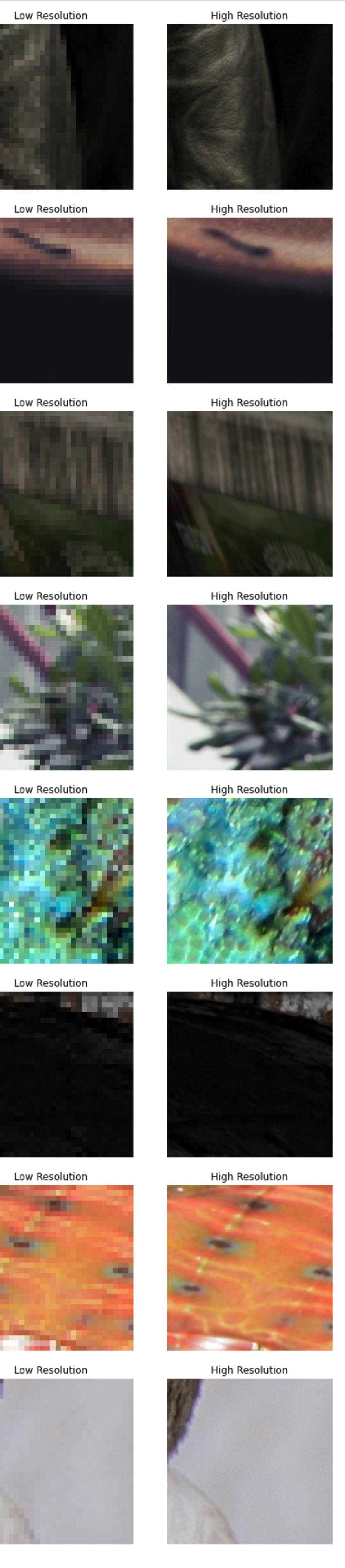
```
(8, 32, 32, 3) (8, 128, 128, 3)
<dtype: 'float32'> <dtype: 'float32'>
tf.Tensor(0.0, shape=(), dtype=float32) tf.Tensor(255.0, shape=(), dtype=float32)
tf.Tensor(0.0, shape=(), dtype=float32) tf.Tensor(255.0, shape=(), dtype=float32)
```

In [ ]:
```
def visualize_samples(images_lists, titles = None, size = (12, 12), masked = False):
    assert len(images_lists) == len(titles)

    cols = len(images_lists)

    for images in zip(*images_lists):
        plt.figure(figsize = size)
        for idx, image in enumerate(images):
            plt.subplot(1, cols, idx + 1)
            plt.imshow(tf.cast(tf.round(tf.clip_by_value(image, 0, 255)), tf.uint8))
            plt.axis('off')
            if titles:
                plt.title(titles[idx])
        plt.show()
```

In [ ]:
```
visualize_samples(images_lists = (lrs[:15], hrs[:15]), titles = ('Low Resolution', 'High Resolution'), size = (
```



Layers for creating models

In [ ]:
```
class Conv2D(layers.Conv2D):
    def __init__(self, kernel_size = 3, padding = 'same', **kwargs):
        super(Conv2D, self).__init__(
            kernel_size = kernel_size,
            padding = padding,
            bias_initializer = tf.keras.initializers.Zeros(),
            **kwargs
        )
```

Modules for creating models

In [ ]:
```
class Conv2DBlock(layers.Layer):
    def __init__(self, filters, batchnorm = True, activate = True, **kwargs):
        super(Conv2DBlock, self).__init__()

        self.conv = Conv2D(filters = filters, **kwargs)
        self.batchnorm = layers.BatchNormalization() if batchnorm else None
        self.activate = layers.PReLU(shared_axes = [1, 2]) if activate else None

    def call(self, inputs):
        x = self.conv(inputs)
        if self.batchnorm:
            x = self.batchnorm(x)
        if self.activate:
            x = self.activate(x)
        return x
```

In [ ]:
```
class ResidualDenseBlock(layers.Layer):
    def __init__(self, filters = 64):
        super(ResidualDenseBlock, self).__init__()

        self.conv1 = Conv2DBlock(filters = filters // 2)
        self.conv2 = Conv2DBlock(filters = filters // 2)
        self.conv3 = Conv2DBlock(filters = filters, activate = False)

    def call(self, inputs):
        x1 = self.conv1(inputs)
        x2 = self.conv2(tf.concat([x1, inputs], 3))
        outputs = self.conv3(tf.concat([x2, x1], 3))

        return outputs + inputs
```

In [ ]:
```
class RRDBlock(layers.Layer):
    def __init__(self, filters, **kwargs):
        super(RRDBlock, self).__init__(**kwargs)

        self.rdb_1 = ResidualDenseBlock(filters)
        self.rdb_2 = ResidualDenseBlock(filters)
        self.rdb_3 = ResidualDenseBlock(filters)

        self.rrdb_inputs_scales = tf.Variable(
            tf.constant(value = 1.0, dtype = tf.float32, shape = [1, 1, 1, filters]),
            name = f'{self.name}_rrdb_inputs_scales',
            trainable = True
        )
        self.rrdb_outputs_scales = tf.Variable(
            tf.constant(value = 0.5, dtype = tf.float32, shape = [1, 1, 1, filters]),
            name = f'{self.name}_rrdb_outputs_scales',
            trainable = True
        )

    def call(self, inputs):
        x1 = self.rdb_1(inputs)
        x2 = self.rdb_2(x1)
        outputs = self.rdb_3(x2)

        return (self.rrdb_inputs_scales * inputs) + (self.rrdb_outputs_scales * outputs)
```

In [ ]:
```
class PixelShuffleUpSampling(layers.Layer):
    def __init__(self, filters, scale, **kwargs):
        super(PixelShuffleUpSampling, self).__init__(**kwargs)

        self.conv1 = Conv2DBlock(filters = filters, batchnorm = False, activate = False)
        self.upsample = layers.Lambda(lambda x: tf.nn.depth_to_space(x, scale))
        self.prelu = layers.PReLU(shared_axes = [1, 2])

    def call(self, x):
        x = self.conv1(x)
        x = self.upsample(x)
        x = self.prelu(x)
        return x
```

Building the models

In [ ]:
```
def Generator():
    lr_image = layers.Input(shape = (None, None, 3))

    spatial_feats = layers.Lambda(lambda x: x / 255.0)(lr_image)
    spatial_feats = Conv2DBlock(filters = GEN_FILTERS, kernel_size = 3, strides = 1, padding = 'same', batchnor
    spatial_feats = Conv2DBlock(filters = GEN_FILTERS, kernel_size = 1, strides = 1, padding = 'valid', batchno

    rrdb1 = RRDBlock(GEN_FILTERS)(spatial_feats)
    rrdb2 = RRDBlock(GEN_FILTERS)(rrdb1)
    rrdb3 = RRDBlock(GEN_FILTERS)(rrdb2)
    rrdb4 = RRDBlock(GEN_FILTERS)(rrdb3)

    upsample1 = PixelShuffleUpSampling(GEN_FILTERS * 4, 2)(rrdb4)
    upsample2 = PixelShuffleUpSampling(GEN_FILTERS * 4, 2)(upsample1)

    x = Conv2DBlock(filters = GEN_FILTERS, batchnorm = False)(upsample2)
    x = Conv2DBlock(filters = 3, kernel_size = 3, activate = False, batchnorm = False)(x)
    x = layers.Activation('tanh')(x)

    sr_image = layers.Lambda(lambda x: (x + 1) * 127.5)(x)

    return Model(inputs = lr_image, outputs = sr_image, name = 'Generator')
```

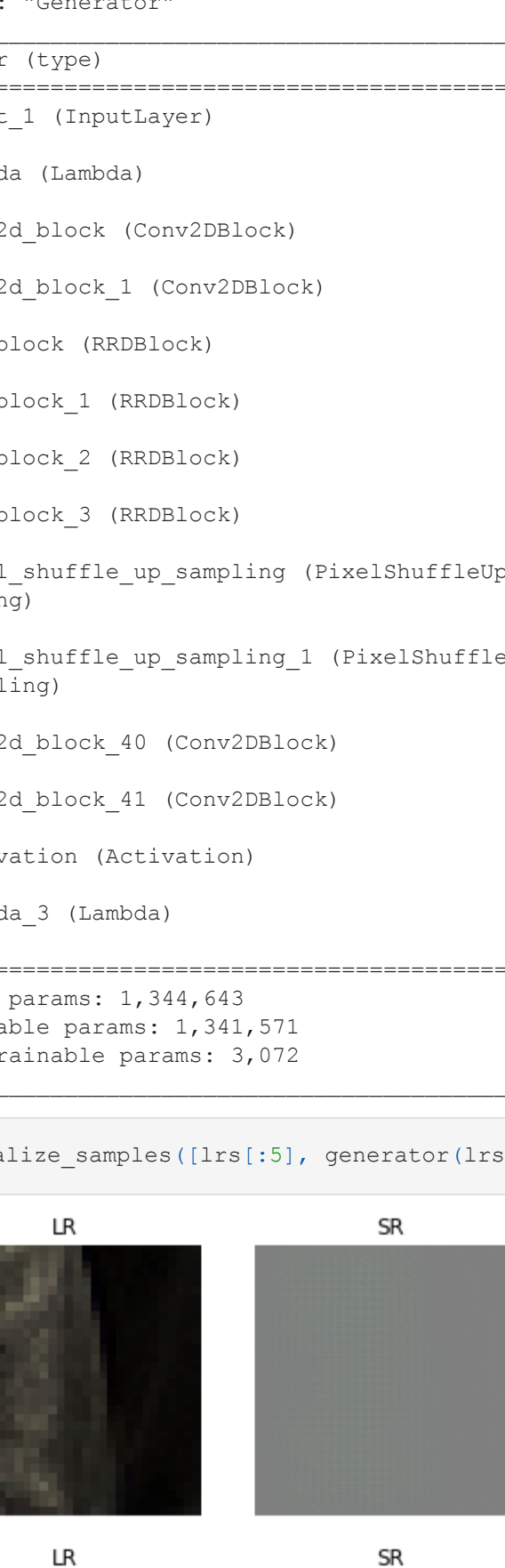
In [ ]:
```
generator = Generator()
generator.summary(100)
```

Model: "Generator"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None, None, 3)]	0
lambda (Lambda)	(None, None, None, 3)	0
conv2d_block (Conv2DBlock)	(None, None, None, 64)	1856
conv2d_block_1 (Conv2DBlock)	(None, None, None, 64)	4224
rrd_block (RRDBlock)	(None, None, None, 64)	251072
rrd_block_1 (RRDBlock)	(None, None, None, 64)	251072
rrd_block_2 (RRDBlock)	(None, None, None, 64)	251072
rrd_block_3 (RRDBlock)	(None, None, None, 64)	251072
pixel_shuffle_up_sampling (PixelShuffleUpSampling)	(None, None, None, 64)	147776
pixel_shuffle_up_sampling_1 (PixelShuffleUpSampling)	(None, None, None, 64)	147776
conv2d_block_40 (Conv2DBlock)	(None, None, None, 64)	36992
conv2d_block_41 (Conv2DBlock)	(None, None, None, 3)	1731
activation (Activation)	(None, None, None, 3)	0
lambda_3 (Lambda)	(None, None, None, 3)	0

Total params: 1,344,643
Trainable params: 1,341,571
Non-trainable params: 3,072

In [ ]:
```
visualize_samples([lrs[:5], generator(lrs[:5])], titles = ['LR', 'SR'], size = (6, 6))
```



In [ ]:
```
def Discriminator():
    hr_image = layers.Input(shape = (HR_SIZE, HR_SIZE, 3))
    x = layers.Lambda(lambda x: x / 127.5 - 1)(hr_image)

    x = Conv2D(kernel_size = 3, filters = DISC_FILTERS // 2)(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS // 2, kernel_size = 3, strides = 2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS, kernel_size = 3)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS, kernel_size = 3, strides = 2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS * 2, kernel_size = 3, strides = 1)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS * 2, kernel_size = 3, strides = 2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS * 4, kernel_size = 3, strides = 1)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = Conv2D(filters = DISC_FILTERS * 4, kernel_size = 3, strides = 2)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(0.2)(x)

    x = layers.Flatten()(x)

    x = layers.Dense(1024)(x)
    x = layers.LeakyReLU(0.2)(x)
    x = layers.Dense(1024)(x)
    x = layers.LeakyReLU(0.2)(x)

    logits = layers.Dense(1)(x)

    return Model(inputs = hr_image, outputs = logits)
```

In [ ]:
```
discriminator = Discriminator()
discriminator.summary(100)
```

Model: "model"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 128, 128, 3)]	0
lambda_4 (Lambda)	(None, 128, 128, 3)	0
conv2d_42 (Conv2D)	(None, 128, 128, 32)	896
leaky_re_lu (LeakyReLU)	(None, 128, 128, 32)	0
conv2d_43 (Conv2D)	(None, 64, 64, 32)	9248
batch_normalization_36 (BatchNormalization)	(None, 64, 64, 32)	128
leaky_re_lu_1 (LeakyReLU)	(None, 64, 64, 32)	0
conv2d_44 (Conv2D)	(None, 64, 64, 64)	18496
batch_normalization_37 (BatchNormalization)	(None, 64, 64, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 64, 64, 64)	0
conv2d_45 (Conv2D)	(None, 32, 32, 64)	36928
batch_normalization_38 (BatchNormalization)	(None, 32, 32, 64)	256
leaky_re_lu_3 (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_46 (Conv2D)	(None, 32, 32, 128)	73856
batch_normalization_39 (BatchNormalization)	(None, 32, 32, 128)	512
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 128)	0
conv2d_47 (Conv2D)	(None, 16, 16, 128)	147584
batch_normalization_40 (BatchNormalization)	(None, 16, 16, 128)	512
leaky_re_lu_5 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_48 (Conv2D)	(None, 16, 16, 256)	295168
batch_normalization_41 (BatchNormalization)	(None, 16, 16, 256)	1024
leaky_re_lu_6 (LeakyReLU)	(None, 16, 16, 256)	0
conv2d_49 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_42 (BatchNormalization)	(None, 8, 8, 256)	1024
leaky_re_lu_7 (LeakyReLU)	(None, 8, 8, 256)	0
flatten (Flatten)	(None, 16384)	0
dense (Dense)	(None, 1024)	16778240
leaky_re_lu_8 (LeakyReLU)	(None, 1024)	0
dense_1 (Dense)	(None, 1024)	1049600
leaky_re_lu_9 (LeakyReLU)	(None, 1024)	0
dense_2 (Dense)	(None, 1)	1025

Total params: 19,004,833
Trainable params: 19,002,977
Non-trainable params: 1,856

Defining training procedures

In [ ]:
```
class PixelLossTraining:
    def setup_pixel_loss(self, pixel_loss):
        if pixel_loss == 'l1':
            self.pixel_loss_type = losses.MeanAbsoluteError()
        elif pixel_loss == 'l2':
            self.pixel_loss_type = losses.MeanSquaredError()

    @tf.function
    def pixel_loss(self, srs, hrs):
        return self.pixel_loss_type(hrs, srs)
```

In [ ]:
```
class VGGContentTraining:
    def setup_content_loss(self, content_loss):
        if content_loss == 'l1':
            self.content_loss_type = losses.MeanAbsoluteError()
        elif content_loss == 'l2':
            self.content_loss_type = losses.MeanSquaredError()

        vgg = applications.VGG19(
            input_shape = (224, 224, 3),
            include_top = False,
            weights = 'imagenet'
        )

        vgg.layers[5].activation = None
        vgg.layers[10].activation = None
        vgg.layers[20].activation = None

        self.feature_extrator = Model(
            inputs = vgg.input,
            outputs = [
                vgg.layers[5].output,
                vgg.layers[10].output,
                vgg.layers[20].output
            ]
        )
        for layer in self.feature_extrator.layers:
            layer.trainable = False

    @tf.function
    def content_loss(self, srs, hrs):
        srs = applications.vgg19.preprocess_input(tf.image.resize(srs, (224, 224)))
        hrs = applications.vgg19.preprocess_input(tf.image.resize(hrs, (224, 224)))

        srs_features = self.feature_extrator(srs)
        hrs_features = self.feature_extrator(hrs)

        loss = 0.0
        for srs_feature, hrs_feature in zip(srs_features, hrs_features):
            loss += self.content_loss_type(hrs_feature / 12.75, srs_feature / 12.75)

        return loss
```

In [ ]:
```
class GramStyleTraining:
    def setup_gram_style_loss(self, style_loss):
        if style_loss == 'l1':
            self.style_loss_type = losses.MeanAbsoluteError()
        elif style_loss == 'l2':
            self.style_loss_type = losses.MeanSquaredError()

        efficientnet = applications.EfficientNetB4(
            input_shape = (224, 224, 3),
            include_top = False,
            weights = 'imagenet'
        )

        self.style_features_extractor = Model(
            inputs = efficientnet.input,
            outputs = [
                # efficientnet.layers[25].output,
                # efficientnet.layers[84].output,
                # efficientnet.layers[143].output,
                efficientnet.layers[320].output,
                # efficientnet.layers[467].output,
            ]
        )
        for layer in self.style_features_extractor.layers:
            layer.trainable = False

    @tf.function
    def gram_matrix(self, features):
        features = tf.transpose(features, (0, 3, 1, 2)) # (-1, C, H, W)
        features_a = tf.reshape(features, (tf.shape(features)[0], tf.shape(features)[1], -1)) # (-1, C, H * W)
        features_b = tf.reshape(features, (tf.shape(features)[0], -1, tf.shape(features)[1])) # (-1, H * W, C)

        return tf.linalg.matmul(features_a, features_b) # (-1, C, C)

    @tf.function
    def gram_style_loss(self, srs, hrs):
        srs = applications.efficientnet.preprocess_input(tf.image.resize(srs, (224, 224)))
        hrs = applications.efficientnet.preprocess_input(tf.image.resize(hrs, (224, 224)))

        srs_features = self.style_features_extractor(srs) # (2, -1, H, W, C)
        hrs_features = self.style_features_extractor(hrs) # (2, -1, H, W, C)

        # style_loss = 0.0
        # for srs_feature, hrs_feature in zip(srs_features, hrs_features):
        srs_gram = self.gram_matrix(srs_features)
        hrs_gram = self.gram_matrix(hrs_features)

        style_loss = self.style_loss_type(hrs_gram, srs_gram)

        return style_loss
```

In [ ]:
```
class AdversarialTraining:
    def setup_adversarial_loss(self, adv_loss):
        self.adv_loss_type = adv_loss
        self.binary_cross_entropy = losses.BinaryCrossentropy(from_logits = True)

    @tf.function
    def gen_adv_loss(self, fake_logits, real_logits = None):
        if self.adv_loss_type == 'gan':
            loss = self.binary_cross_entropy(tf.ones_like(fake_logits), fake_logits)

        elif self.adv_loss_type == 'ragan':
            real_loss = self.binary_cross_entropy(tf.ones_like(fake_logits), fake_logits - tf.reduce_mean(real_
            fake_loss = self.binary_cross_entropy(tf.zeros_like(real_logits), real_logits - tf.reduce_mean(fake
            loss = real_loss + fake_loss

        return loss

    @tf.function
    def disc_adv_loss(self, fake_logits, real_logits):
        if self.adv_loss_type == 'gan':
            real_loss = self.binary_cross_entropy(tf.ones_like(real_logits), real_logits)
            fake_loss = self.binary_cross_entropy(tf.zeros_like(fake_logits), fake_logits)

        elif self.adv_loss_type == 'ragan':
            real_loss = self.binary_cross_entropy(tf.ones_like(real_logits), real_logits - tf.reduce_mean(fake_
            fake_loss = self.binary_cross_entropy(tf.zeros_like(fake_logits), fake_logits - tf.reduce_mean(real

        return real_loss + fake_loss
```

Defining SRGAN Model

In [ ]:
```
class SRGAN(
        Model,
        PixelLossTraining,
        GramStyleTraining,
        VGGContentTraining,
        AdversarialTraining
    ):
    def __init__(
        self,
        generator,
        discriminator,
    ):
        super(SRGAN, self).__init__(self, dynamic = True)

        self.generator = generator
        self.discriminator = discriminator

    def compile(
        self,

        generator_optimizer,
        discriminator_optimizer,

        perceptual_finetune,

        pixel_loss,
        style_loss,
        content_loss,
        adv_loss,

        loss_weights,
    ):
        super(SRGAN, self).compile()

        self.generator.optimizer = generator_optimizer
        self.discriminator.optimizer = discriminator_optimizer

        self.perceptual_finetune = perceptual_finetune

        self.setup_pixel_loss(pixel_loss)
        # self.setup_gram_style_loss(style_loss)
        self.setup_content_loss(content_loss)
        self.setup_adversarial_loss(adv_loss)

        if self.perceptual_finetune:
            self.loss_weights = loss_weights

    def train_step(self, batch):
        self.lrs = batch[0]
        self.hrs = batch[1]

        if self.perceptual_finetune:
            # [================== Training Discriminator ==================]

            with tf.GradientTape() as disc_tape, tf.GradientTape() as gen_tape:
                self.srs = self.generator(self.lrs, training = True)

                real_logits = self.discriminator(self.hrs, training = True)
                fake_logits = self.discriminator(self.srs, training = True)

                content_loss = self.loss_weights['content_loss'] * self.content_loss(self.srs, self.hrs)
                gen_adv_loss = self.loss_weights['adv_loss'] * self.gen_adv_loss(fake_logits, real_logits)
                perceptual_loss = content_loss + gen_adv_loss

                # style_loss = self.loss_weights['style_loss'] * self.gram_style_loss(self.srs, self.hrs)

                gen_loss = perceptual_loss

                disc_adv_loss = self.disc_adv_loss(fake_logits, real_logits)

            discriminator_gradients = disc_tape.gradient(disc_adv_loss, self.discriminator.trainable_variables)
            generator_gradients = gen_tape.gradient(gen_loss, self.generator.trainable_variables)

            self.discriminator.optimizer.apply_gradients(zip(discriminator_gradients, self.discriminator.traina
            self.generator.optimizer.apply_gradients(zip(generator_gradients, self.generator.trainable_variable

            return {
                'Perceptual Loss': perceptual_loss,
                # 'Style Loss': style_loss,
                'Generator Adv Loss': gen_adv_loss,
                'Discriminator Adv Loss': disc_adv_loss,
            }

        else:
            with tf.GradientTape() as gen_tape:
                self.srs = self.generator(self.lrs, training = True)

                pixel_loss = self.pixel_loss(self.srs, self.hrs)

            generator_gradients = gen_tape.gradient(pixel_loss, self.generator.trainable_variables)
            self.generator.optimizer.apply_gradients(zip(generator_gradients, self.generator.trainable_variable

            return {
                'Pixel Loss': pixel_loss,
            }
```

Checkpoint Callback

```python
class CheckpointCallback(tf.keras.callbacks.Callback):
    def __init__(self, checkpoint_dir, resume = False, epoch_step = 1):
        super(CheckpointCallback, self).__init__()

        self.checkpoint_dir = checkpoint_dir
        self.resume = resume
        self.epoch_step = epoch_step

    def setup_checkpoint(self, *args, **kwargs):
        self.checkpoint = tf.train.Checkpoint(
            generator = self.model.generator,
            discriminator = self.model.discriminator,
            generator_optimizer = self.model.generator.optimizer,
            discriminator_optimizer = self.model.discriminator.optimizer
        )
        self.manager = tf.train.CheckpointManager(
            self.checkpoint,
            directory = self.checkpoint_dir,
            checkpoint_name = 'SRGAN',
            max_to_keep = 1
        )

        if self.resume:
            self.load_checkpoint()
        else:
            print('Starting training from scratch...\n')

    def on_batch_end(self, batch, *args, **kwargs):
        if (batch + 1) % int(self.epoch_step * len(train_data)) == 0:
            print(f"\n\nCheckpoint saved to {self.manager.save()}\n")

    def load_checkpoint(self):
        if self.manager.latest_checkpoint:
            self.checkpoint.restore(self.manager.latest_checkpoint)
            print(f"Checkpoint restored from '{self.manager.latest_checkpoint}'\n")
        else:
            print("No checkpoints found, initializing from scratch...\n")

    def set_lr(self, lr, beta_1 = 0.9):
        print(f'Continuing with learning rate: {lr}')
        self.model.generator.optimizer.beta_1 = beta_1
        self.model.generator.optimizer.learning_rate = lr
        self.model.discriminator.optimizer.beta_1 = beta_1
        self.model.discriminator.optimizer.learning_rate = lr
```

Optimization Progress Callback

```python
class ProgressCallback(tf.keras.callbacks.Callback):
    def __init__(self, logs_step, generator_step):
        super(ProgressCallback, self).__init__()

        self.logs_step = logs_step
        self.generator_step = generator_step

    def on_batch_end(self, batch, logs, **kwargs):
        if (batch + 1) % int(self.generator_step * len(train_data)) == 0:
            if self.model.perceptual_finetune:
                visualize_samples(
                    images_lists = (self.model.lrs[:3], self.model.srs[:3], self.model.hrs[:3]),
                    titles = ('Low Resolution', 'Predicted Enhanced', 'High Resolution'),
                    size = (11, 11)
                )
            else:
                visualize_samples(
                    images_lists = (self.model.lrs[:3], self.model.srs[:3]),
                    titles = ('Low Resolution', 'Predicted Enhanced'),
                    size = (7, 7)
                )
```

Optimization Config Values

```python
EPOCHS = 100
LR = 0.00002
BETA_1 = 0.8
BETA_2 = 0.999

PERCEPTUAL_FINETUNE = True

PIXEL_LOSS = 'l1'
STYLE_LOSS = 'l1'
CONTENT_LOSS = 'l1'
ADV_LOSS = 'ragan'

LOSS_WEIGHTS = {'content_loss': 1.0, 'adv_loss': 0.09, 'style_loss': 1.0}

CHECKPOINT_DIR = os.path.join('drive', 'MyDrive', 'Model-Checkpoints', 'Super Resolution')
```

Initializing Models

```python
generator_optimizer = optimizers.Adam(
    learning_rate = LR,
    beta_1 = BETA_1,
    beta_2 = BETA_2
)
discriminator_optimizer = optimizers.Adam(
    learning_rate = LR,
    beta_1 = BETA_1,
    beta_2 = BETA_2
)

srgan = SRGAN(generator, discriminator)
srgan.compile(
    generator_optimizer = generator_optimizer,
    discriminator_optimizer = discriminator_optimizer,

    perceptual_finetune = PERCEPTUAL_FINETUNE,
    pixel_loss = PIXEL_LOSS,
    style_loss = STYLE_LOSS,
    content_loss = CONTENT_LOSS,
    adv_loss = ADV_LOSS,

    loss_weights = LOSS_WEIGHTS
)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_o
rdering_tf_kernels_notop.h5
80142336/80134624 [==============================] - 1s 0us/step
80150528/80134624 [==============================] - 1s 0us/step
```

Setting up checkpoint callback

```python
ckpt_callback = CheckpointCallback(
    checkpoint_dir = CHECKPOINT_DIR,
    resume = True,
    epoch_step = 4
)
ckpt_callback.set_model(srgan)
ckpt_callback.setup_checkpoint(srgan)
ckpt_callback.set_lr(0.0002, BETA_1)
```

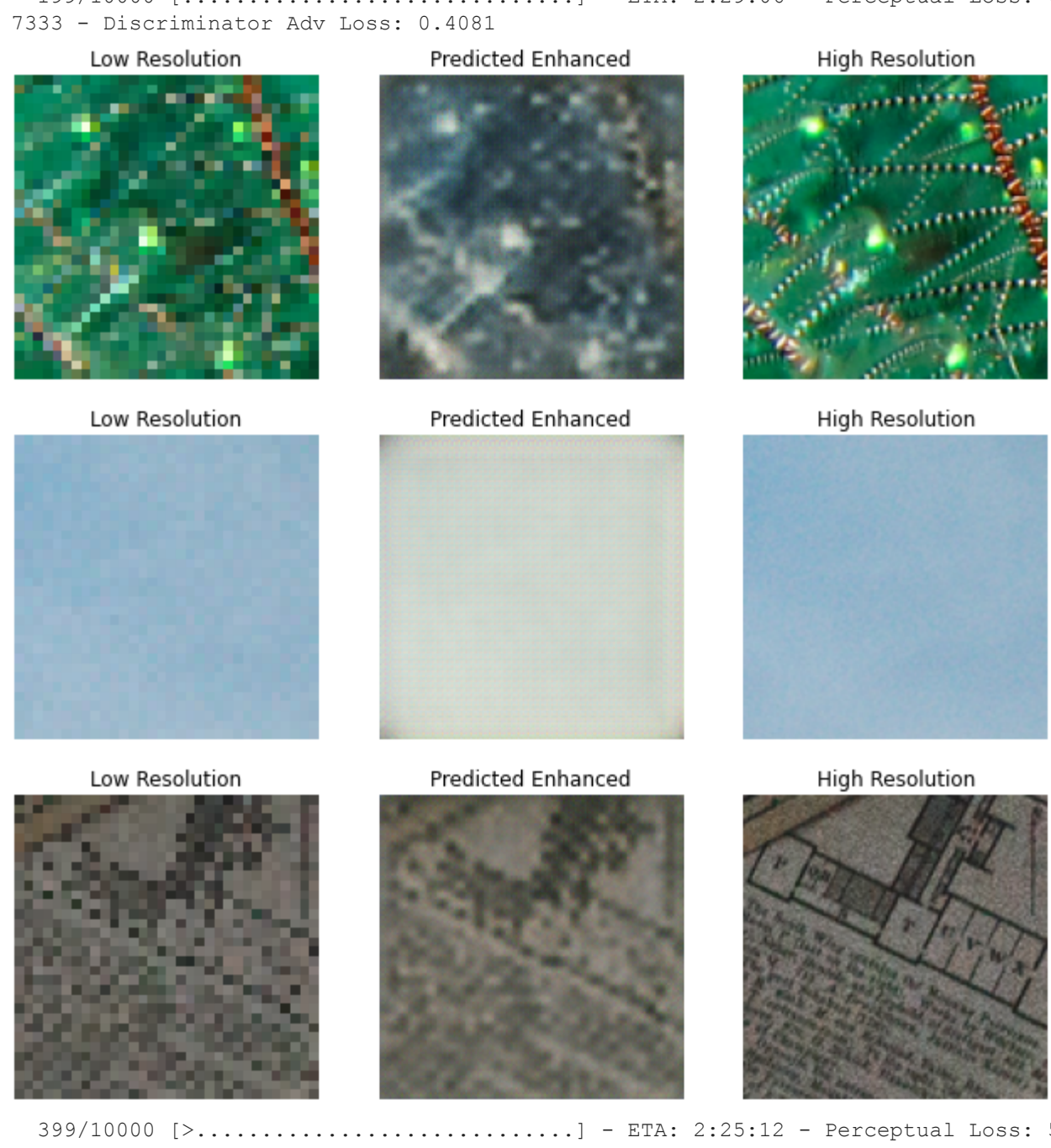
```
No checkpoints found, initializing from scratch...

Continuing with learning rate: 0.0002
```

Training the Model

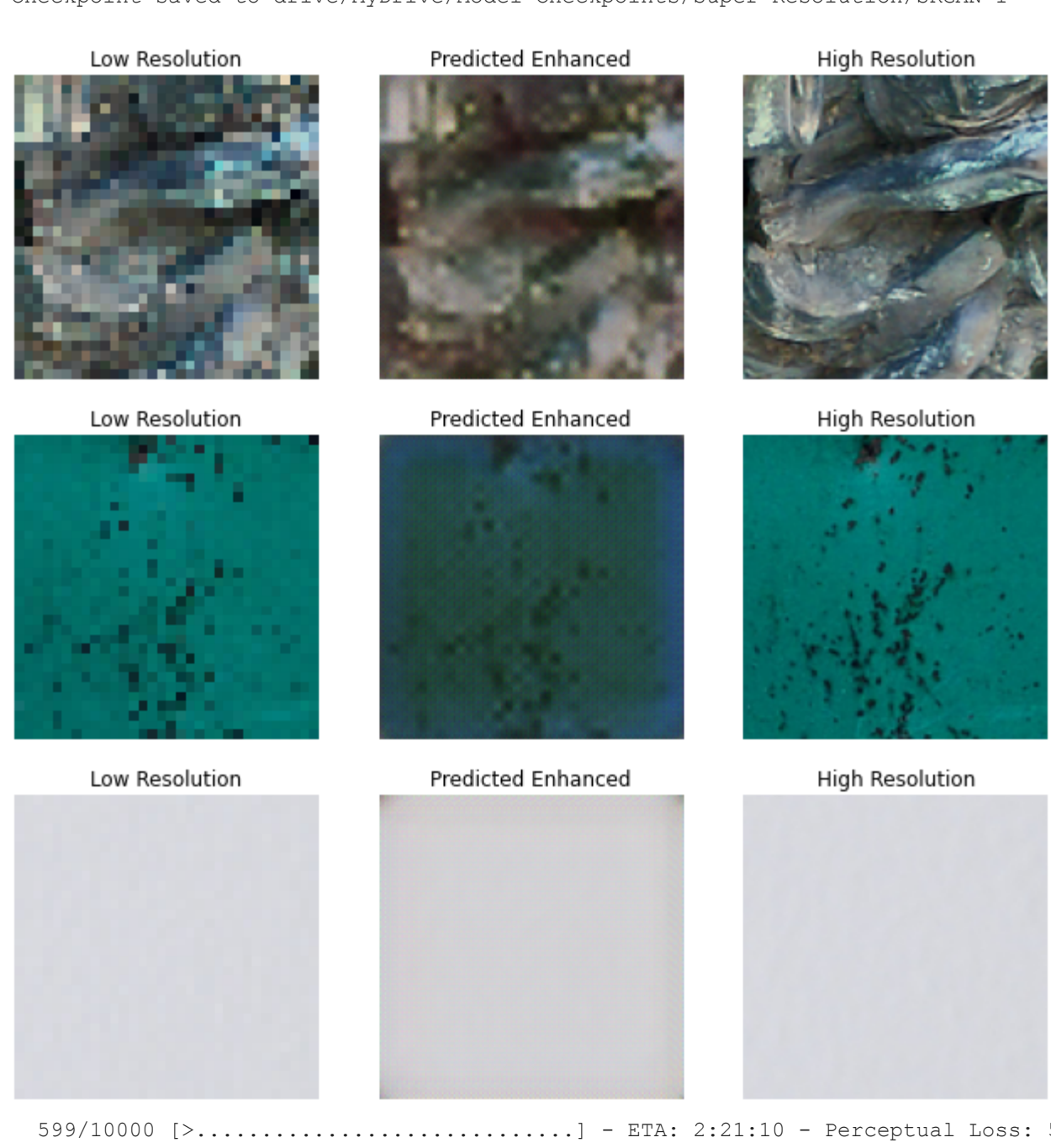
```python
srgan.fit(
    train_data.repeat(EPOCHS // 10),
    epochs = 10,
    callbacks = [
        ckpt_callback,
        ProgressCallback(
            logs_step = 0.2,
            generator_step = 2
        )
    ]
)
```

```
Epoch 1/10
  199/10000 [..............................] - ETA: 2:29:06 - Perceptual Loss: 60.9603 - Generator Adv Loss: 1.
7333 - Discriminator Adv Loss: 0.4081
```

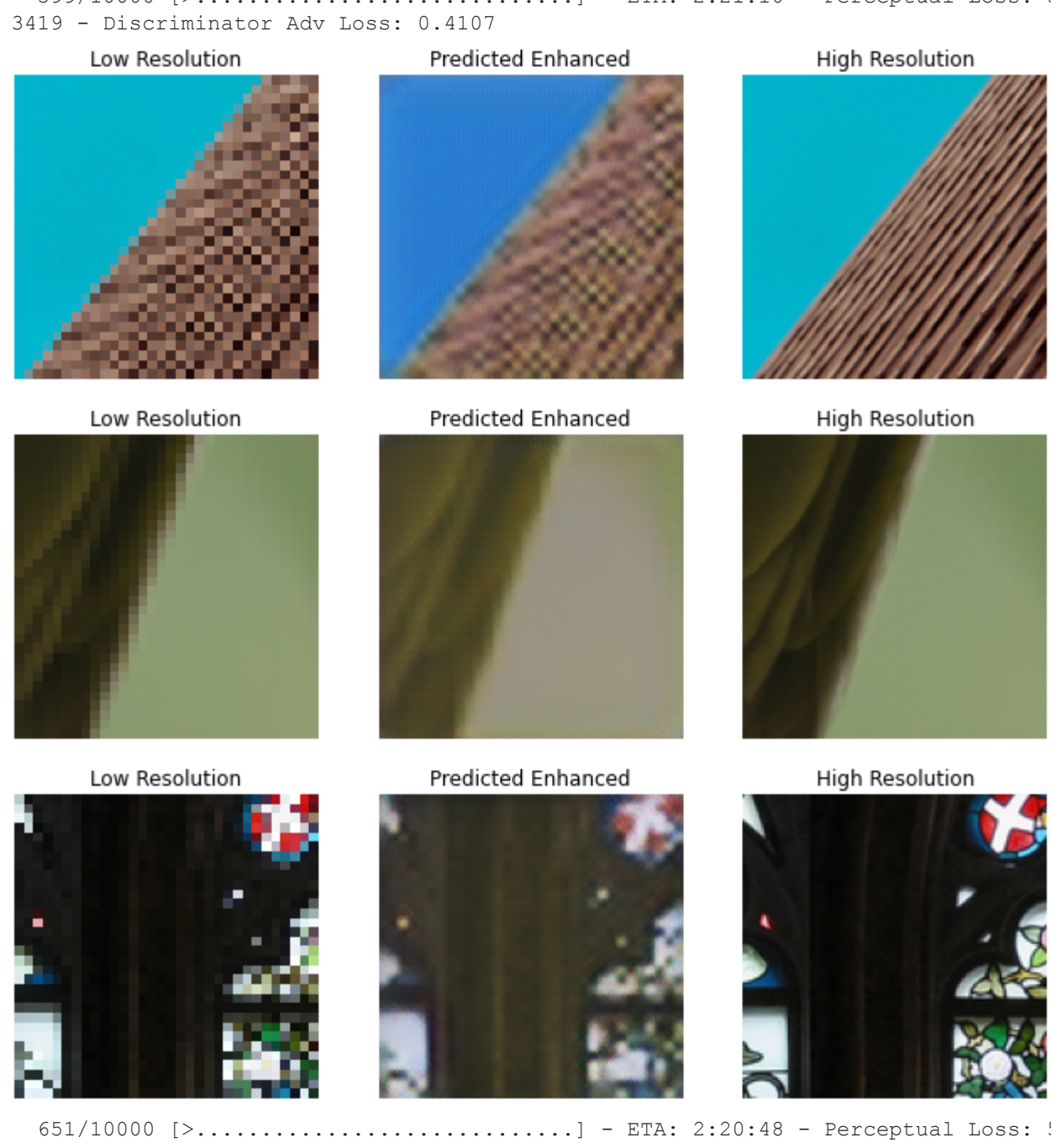



```
  399/10000 [>.............................] - ETA: 2:25:12 - Perceptual Loss: 55.5417 - Generator Adv Loss: 1.
5325 - Discriminator Adv Loss: 0.4165

Checkpoint saved to drive/MyDrive/Model-Checkpoints/Super Resolution/SRGAN-1
```




```
  599/10000 [>.............................] - ETA: 2:21:10 - Perceptual Loss: 52.5495 - Generator Adv Loss: 1.
3419 - Discriminator Adv Loss: 0.4107
```




```
  651/10000 [>.............................] - ETA: 2:20:48 - Perceptual Loss: 52.0914 - Generator Adv Loss: 1.
3464 - Discriminator Adv Loss: 0.3930
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-32-bc0af67cc0f3> in <module>()
      6         ProgressCallback(
      7             logs_step = 0.2,
----> 8             generator_step = 2
      9         )
     10     ]

/usr/local/lib/python3.7/dist-packages/keras/utils/traceback_utils.py in error_handler(*args, **kwargs)
     62     filtered_tb = None
     63     try:
---> 64       return fn(*args, **kwargs)
     65     except Exception as e:  # pylint: disable=broad-except
     66       filtered_tb = _process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.7/dist-packages/keras/engine/training.py in fit(self, x, y, batch_size, epochs, verbose,
 callbacks, validation_split, validation_data, shuffle, class_weight, sample_weight, initial_epoch, steps_per_e
poch, validation_steps, validation_batch_size, validation_freq, max_queue_size, workers, use_multiprocessing)
   1382                 _r=1):
   1383               callbacks.on_train_batch_begin(step)
-> 1384               tmp_logs = self.train_function(iterator)
   1385               if data_handler.should_sync:
   1386                 context.async_wait()

/usr/local/lib/python3.7/dist-packages/keras/engine/training.py in train_function(iterator)
   1019       def train_function(iterator):
   1020         """Runs a training execution with a single step."""
-> 1021         return step_function(self, iterator)
   1022
   1023       if not self.run_eagerly:

/usr/local/lib/python3.7/dist-packages/keras/engine/training.py in step_function(model, iterator)
   1008             run_step, jit_compile=True, experimental_relax_shapes=True)
   1009       data = next(iterator)
-> 1010       outputs = model.distribute_strategy.run(run_step, args=(data,))
   1011       outputs = reduce_per_replica(
   1012           outputs, self.distribute_strategy, reduction='first')

/usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py in run(***failed resolvin
g arguments***)
   1310       fn = autograph.tf_convert(
   1311           fn, autograph_ctx.control_status_ctx(), convert_by_default=False)
-> 1312       return self._extended.call_for_each_replica(fn, args=args, kwargs=kwargs)
   1313
   1314   def reduce(self, reduce_op, value, axis):

/usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py in call_for_each_replica
(self, fn, args, kwargs)
   2886       kwargs = {}
   2887     with self._container_strategy().scope():
-> 2888       return self._call_for_each_replica(fn, args, kwargs)
   2889
   2890   def _call_for_each_replica(self, fn, args, kwargs):

/usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py in _call_for_each_replica
(self, fn, args, kwargs)
   3687   def _call_for_each_replica(self, fn, args, kwargs):
   3688     with ReplicaContext(self._container_strategy(), replica_id_in_sync_group=0):
-> 3689       return fn(*args, **kwargs)
   3690
   3691   def _reduce_to(self, reduce_op, value, destinations, options):

/usr/local/lib/python3.7/dist-packages/tensorflow/python/autograph/impl/api.py in wrapper(*args, **kwargs)
    593   def wrapper(*args, **kwargs):
    594     with ag_ctx.ControlStatusCtx(status=ag_ctx.Status.UNSPECIFIED):
--> 595       return func(*args, **kwargs)
    596
    597   if inspect.isfunction(func) or inspect.ismethod(func):

/usr/local/lib/python3.7/dist-packages/keras/engine/training.py in run_step(data)
    998
    999       def run_step(data):
-> 1000         outputs = model.train_step(data)
   1001         # Ensure counter is updated only if `train_step` succeeds.
   1002         with tf.control_dependencies(_minimum_control_deps(outputs)):

<ipython-input-22-a06f9a1c5276> in train_step(self, batch)
     69                 disc_adv_loss = self.disc_adv_loss(fake_logits, real_logits)
     70
---> 71             discriminator_gradients = disc_tape.gradient(disc_adv_loss, self.discriminator.trainable_va
riables)
     72             generator_gradients = gen_tape.gradient(gen_loss, self.generator.trainable_variables)
     73

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/backprop.py in gradient(self, target, sources, o
utput_gradients, unconnected_gradients)
   1085         output_gradients=output_gradients,
   1086         sources_raw=flat_sources_raw,
-> 1087         unconnected_gradients=unconnected_gradients)
   1088
   1089     if not self._persistent:

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/imperative_grad.py in imperative_grad(tape, targ
et, sources, output_gradients, sources_raw, unconnected_gradients)
     71       output_gradients,
     72       sources_raw,
---> 73       compat.as_str(unconnected_gradients.value))

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/backprop.py in _gradient_function(op_name, attr_
tuple, num_inputs, inputs, outputs, out_grads, skip_input_indices, forward_pass_name_scope)
    154       gradient_name_scope += forward_pass_name_scope + "/"
    155     with ops.name_scope(gradient_name_scope):
--> 156       return grad_fn(mock_op, *out_grads)
    157   else:
    158     return grad_fn(mock_op, *out_grads)

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/nn_grad.py in _Conv2DGrad(op, grad)
    594           explicit_paddings=explicit_paddings,
    595           use_cudnn_on_gpu=use_cudnn_on_gpu,
--> 596           data_format=data_format)
    597   ]
    598

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/gen_nn_ops.py in conv2d_backprop_filter(input, fil
ter_sizes, out_backprop, strides, padding, use_cudnn_on_gpu, explicit_paddings, data_format, dilations, name)
   1082         "strides", strides, "use_cudnn_on_gpu", use_cudnn_on_gpu, "padding",
   1083         padding, "explicit_paddings", explicit_paddings, "data_format",
-> 1084         data_format, "dilations", dilations)
   1085       return _result
   1086     except _core._NotOkStatusException as e:

KeyboardInterrupt:
```

Testing the model

```python
def enhance_image(lr_image = None, path = None, output_path = None, visualize = True, size = (20, 16)):
    assert any([lr_image is not None, path])
    if path:
        lr_image = tf.image.decode_jpeg(tf.io.read_file(f"{path}"), channels = 3)

    sr_image = srgan.generator(tf.expand_dims(lr_image, 0), training = False)[0]
    sr_image = tf.clip_by_value(sr_image, 0, 255)
    sr_image = tf.round(sr_image)
    sr_image = tf.cast(sr_image, tf.uint8)

    if visualize:
        visualize_samples(images_lists = [[lr_image], [sr_image]], titles = ['LR Image', 'SR_Image'], size = si

    if output_path:
        tf.io.write_file(output_path, tf.image.encode_jpeg(sr_image))
```

```python
for idx, (lr, _) in enumerate(tfds.load(f'div2k/bicubic_x{SCALE}', split = 'validation', shuffle_files = True).
    lr_x = tf.random.uniform(maxval = int(lr.shape[1] // 2), shape = (), dtype = tf.int32)
    lr_y = tf.random.uniform(maxval = int(lr.shape[0] // 2), shape = (), dtype = tf.int32)
    lr = lr[lr_x: lr_x + 100, lr_y: lr_y + 100, :]
    sr = enhance_image(lr_image = lr)
```

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image

LR Image

SR_Image