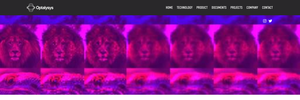
AI super resolution using an optical computer

[18 minutes read]

In this article we will discuss super resolution, an AI technique that uses deep neural networks to increase the resolution of images and videos.



(Image credit: Dave Lonsdale, Flickr, Flickr CC)

Gone are the days when the above sketch could be of the most sophisticated computer vision technology. Today, we ready cell phones and operate low resolution deep learning models on the go, in real time.

Yet even more fundamentally, Fourier optical computing can be used to accelerate this AI processing by reducing the separable computational complexity of convolution operations from O(n^2) to linear. (So cool!) More on this later.

Article contents

- Offline super resolution
- How super resolution works
- CNNs for super resolution: upsampling techniques
- PyTorch implementation of super resolution
- Experiments with different loss functions
- Optical implementation of super resolution

What is super resolution?

Simply put, super resolution generates better, higher resolution versions of images. Cell phones can take low resolution images (or frames of low resolution video) and output high resolution data that looks good. Better than bilinear or bicubic interpolation. Better to the human eye.

This almost seems like magic, and for good reason: somehow these networks are able to take grainy, pixelated inputs and transform them into high resolution, great looking images.

All this can be achieved without explicitly using any of the traditional computer graphics techniques such as anti-aliasing, smoothing, sharpening, edge detection, etc. Instead, these networks are trained on pairs of low and high resolution images, and the low resolution networks are able to infer a statistical image distortion by using prior knowledge and patterns learned from examples.

How super resolution works

A super resolution network takes images (deep neural networks optimized for Cc Convolutional neural networks (CNNs) have been a powerful tool for visual tasks. Image classification, for example, can be forced into fixed-size classification, and the majority of which is Yann Lecun's LeNet CNNs for handwritten digits, allowing them to be processed by fully-connected layers.

Background: CNNs for classification

Classification CNNs typically have a structure like this:

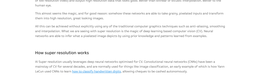


Image of a typical CNN used for classification. Large images are pooled down to smaller images sequentially, eventually the smaller images connect to a dense layer for classification.

Typically, images are fed into the network and convolved with many filters, the filtered images are combined to make up the output of a hidden layer. This process repeat. The last layer is usually the same size as the input.

Convolutional layers can be thought of as sliding a window across the image and performing a matrix multiplication at each window. Once the data has been pooled to a reasonable size (sometimes to a single pixel), it can be fed into dense or fully-connected layers for classification.

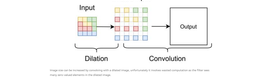
Background: CNNs for other systems

More modern CNN architectures include autoencoders (driving robotic surgery and medical screening (these applications go beyond image classification, into systems control and semantic segmentation). Fully-convolutional networks (such as Facebook's end-to-end trainable CC, where both the CNN inputs and outputs are images. Such systems are able to manipulate a pair of images, non-invasively transporting the flow of one into the flow of another.

A Faceswap recipe CNN may have a structure like this:

In an image-image CNN, dimensionality of the input is not necessarily reduced. Sequential convolution layers can be used without pooling.

Or like this:



Typical U-Net style CNN, where the intermediate layer output is downsampled (pooled) first, before being exponentially upsampled to an increased size. The reduction in size is usually accompanied by an increase in the channel dimension to stop too much information being thrown away in the middle.

CNNs for super resolution

In a similar way, super resolution networks take an image and output another. The only difference from the above architectures is that the input image and output image are not the same size. Rather the output should be of a higher resolution, while still faithfully depicting the original scene.

If we denote the output of the CNN to be of a higher resolution than the input, then we require some form of non-standard CNN layer, something which is the opposite of a typical convolution. It upsamples.

Upsampling layers can be implemented in various ways, including as a convolution. Image resolution should increase, therefore it is possible to also reduce the channel count. This can be achieved by reshaping. For example, a 1ffc channel 1024x768 input could be reshaped into a 4 channel 2048x1536 output, reducing the channel count by a factor of 4. This is one implementation of a transposed CNN layer (with stride factor equal to the upscaling factor). Unfortunately, this computer prohibitively.

One way to think of this is to take the big leftt pixel of the first 4 channels, and assemble a 2x2 pixel patch. Then do the same thing with the 2nd pixel, and so on. This is a sub-pixel convolution. This trick can be repeated for all pixel locations and channels set allocation.

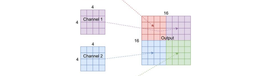
Image credit: Vincent Dumoulin and Francesco Visin

Multi-channel reshape upsampling [aka sub-pixel convolution]

Channel upsampling takes a multi-channel image (such as the output from a convolution layer) and reshapes the data into fewer channels but with larger height and width.

In a classification CNN, there are usually sequential pooling layers, designed to reduce the size of the internal data representation and ultimately the computational complexity. In order to make sure unsupervisable information is not being destroyed, convolution layers or blocks typically increase the channel count. For instance a 32-channel 16x16 input may be converted into a 64-channel 8x8 output.

Upsampling layers can be used in a similar way. Instead, image resolution should increase, therefore it is possible to also reduce the channel count. This can be achieved by reshaping. For example, a 16-channel 1024x768 input could be reshaped into a 4 channel 2048x1536 output, reducing the channel count by a factor of 4. This is one implementation of a transposed CNN layer (with stride factor equal to the upscaling factor). Unfortunately, this computer prohibitively.



Note multiple channels can be combined to create a larger single channel of larger resolution. In this example 4 input channels of size 4x4 are upsampled into a single 8x8 output channel.

We could of course just reshape the output tensor naively, however this might cause problems, because CNNs preserve the spatial information of their inputs, so it makes sense to keep pixels in the same spatial location together.

This keeps nearby pixels together, which is reasonable for spatial upsampling. The problem here is that the global representation in each of the input channels is incorrectly combined into a local spatial location.

When to upsample

One final consideration is when the upsampling should take place. There is no 'correct' solution, just a few different ideas with varying results.

- Upsampling in the final layer. This is the most simple idea however, the authors of Deep Learning for Image Super-resolution: A Survey note that this kind of layer operates factors such as all this method is very slow.
- Upsample gradually throughout the network, similar to the sequential pooling throughout classification CNNs.
- Upsample and denoise recipes throughout the network, e.g. U-Net architecture.

PyTorch implementation of super resolution

Data & hardware

Data for super resolution experiments is abundant. Images need not be classified or labelled in any way, the only prerequisite is that training images should be at least as big as the resolution we wish to upsample to.

We used a python web scraper that downloads images returned by Google image search when fed keywords like 'fox':

```
# ip address lists and headers
ip_list = []
header = {}
import bs4
import requests
import re
import urllib2
import os

def bing_image_urls(search, ext=None, stream=False, max_results=None, pages=None, timeout=60.0):
    """ Returns a list of image urls for a bing image search.
    If stream=True, returns a generator.
    Timeout is overridden in seconds.
    If ext=None, also searches for .png and .gif images.
    """
    ua = urllib2.HTTPCookieProcessor(urllib2.CookieJar())
    urllib2.install_opener(ua)
    timeout_ = timeout
    if ext == None:
        ext = ''
    stream = False
    count = 0
    while True:
        response = bing_page(search, ext, pages)
        if not response:
            raise StopIteration()
        found = re.findall('"murl":"(.*?)",', response.read().decode('ascii', errors='replace'))
        if not found:
            raise StopIteration()
        for u in found:
            if max_results and count >= max_results:
                raise StopIteration()
            yield u
            count += 1
        if not stream:
            timeout = timeout + timeout_
        if pages is None:
            pages = (pages or 0) + 1
        if max_results:
            pass
        else:
            raise StopIteration()
def bing_page(search, ext, page):
    ua = urllib2.urlopen('http://www.bing.com/images/search?q=%s+filetype%3A%s&first=%s' % (search, ext, (page or 1) * 35), timeout=60.0)
    return ua

if __name__ == '__main__':
    search = 'fox'
    ext = 'jpg'
    pages = None
    max_results = None
    stream = False
    timeout = 60.0
    verbose = False
    path = None
    from sys import *
    from os.path import *
    from os import *
    from time import ctime
    from errno import EEXIST
    from os.path import join, basename
    from os.makedirs import makedirs, OSError
    from os.stat import st_mtime
    from os.utime import utime
    from os.sys.stdout import write
    from os.sys.stderr import write
    if len(argv) > 1:
        search = argv[1]
    if len(argv) > 2:
        ext = argv[2]
    if len(argv) > 3:
        pages = int(argv[3])
    if len(argv) > 4:
        max_results = int(argv[4])
    if len(argv) > 5:
        stream = bool(argv[5])
    if len(argv) > 6:
        timeout = float(argv[6])
    if len(argv) > 7:
        verbose = bool(argv[7])
    if len(argv) > 8:
        path = argv[8]
    if not search:
        print("Error: no search term.")
        print("Usage: bing_image_urls.py [search] [ext] [pages] [max_results] [stream] [timeout] [verbose] [path]")
        exit(2)
    if stream:
        print("Error: not implemented yet.")
    if not path:
        path = join('.', search)
    if verbose:
        print("Using path %s." % path)
    if not exists(path):
        makedirs(path)
    elif not isdir(path):
        print("Error: %s is a file." % path, file=stderr)
        exit(1)
    if verbose:
        print("Searching Bing for %s filetype:%s" % (search, ext))
    try:
        ln = 0
        if verbose:
            print("Page offset: %s" % pages)
        for u in bing_image_urls(search, ext=ext, pages=pages, max_results=max_results, stream=stream, timeout=timeout):
            ln += 1
            if verbose:
                print("found image %s" % u)
            else:
                write('.')
            try:
                if verbose:
                    print("Opening %s" % u)
                r = urlopen(u, timeout=timeout)
                f = open(join(path, basename(u)), 'wb')
                write(f, r.read())
                f.close()
                if verbose:
                    print("Wrote to %s" % f.name)
                else:
                    write('+')
                utime(f.name, (-1, int(time())))
                if verbose:
                    print("Updated timestamp.")
                else:
                    write('+')
                pass
            except OSError, e:
                if e.errno == EEXIST:
                    if verbose:
                        print("%s: %s" % (e.filename, e.strerror), file=stderr)
                    else:
                        write('-')
                    pass
                if verbose:
                    print("%s: %s" % (e.filename, e.strerror), file=stderr)
                else:
                    write('-')
                pass
            except IOError, e:
                if verbose:
                    print("%s: %s" % (e.filename, e.strerror), file=stderr)
                else:
                    write('-')
                pass
        if ln == 0:
            if verbose:
                print("Error: no results.")
            else:
                write('!')
            exit(3)
        if verbose:
            print("Done.")
        else:
            write('\n')
        exit(0)
    except KeyboardInterrupt, e:
        if verbose:
            print("Interrupted.")
        else:
            write('\n')
        exit(4)
```

bing_image_urls.py

Image large enough were downloaded, cropped square and resized to our target upscaled resolution (256). This process was repeated until we had a unique dataset of around 16,000 images. The images were split into 70% training and 30% validation subsets. PyTorch dataloaders were used to load the datasets.

Training setup

During training and validation, the 3 channel, 256^2 sized images in the dataset were used as the ground truth values. Inputs to the network were created by downscaling the ground truth images to 3 channel 32^2 using the nearest neighbour algorithm. Electronic training took place on an Nvidia Quadro P6000.

We learned our convolutional filters to be performed by Sandycat optical chip, so we used our PyTorch layer to interface with the optical computer. Although this was slower than using a GPU, it allowed us to simulate the performance of our optical layer. Although hyperparameters were chosen, the optical network could be called by passing 'optical' flag instead of a non-linear activation.

Our upsampling technique

We wrote code that defines an upsample block of varying channel depth and convolutional layer count. The block could then be recursively called to create a network of any depth and complexity. Our upsampling technique was inspired by sub-pixel convolution and ESPCN and to and network separable factors. The block uses pixel-to-channel reshape (aka sub-pixel convolution) followed by a 3-layer convolution architecture.

Avoiding checkerboarding

Upsampling generally can suffer from 'checkerboarding', clearly visible in this image below:

Example of checkerboarding
(Original Image Source)

This paper discusses it in depth, and provides a principled method to avoid it. As we used the so-called 'nearest-convolution' upsampling channel, we precomputed this by initializing the kernels to mimicking the weights and biases of the convolutional kernels which contribute to the same output pixelset to the same value.

Padding

A further generic issue with upsampling CNNs can occur at the border of the output image. Generally in CNNs, padding is used to keep the height and width of layer inputs and outputs consistent. Zero-padding is the traditional choice, however this effectively would mean that we are outsmarting pixels with black border, which would lead to visual artefacts in the upscaled image. For this reason we also implemented reflect padding so that the border would be made up of sensible values.

Upsample block PyTorch code

Here is the code for the upsample block:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.ops.misc import ConvTranspose

# kernels 3 kernels mean no upsampling 0.0001, 0.0, 0.0 * 1.0, 0.0, 0.0 0.0
def get_kernels(ks, verbose):
    ''' ks - kernel size
    ret - 3D list of shape [3][ks][ks]
    '''
    ret = [[[0.0 for j in range(ks)] for i in range(ks)] for c in range(3)]
    if ks == 3:
        ret = [[[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.0]]]
    elif ks == 5:
        ret = [[[0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0]]]
    elif ks == 7:
        ret = [[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]]
    elif ks == 9:
        ret = [[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]]
    if verbose:
        print("Kernels: %s" % ret)
    return ret

# convolution kernel that mimics a bilinear upsampling
# ks - kernel size, ic - input channels, oc - output channels, verbose - verbosity
def get_bilinear_upsampling_weights(ks, ic, oc, verbose):
    ''' ks - kernel size, ic - input channels, oc - output channels, verbose - verbosity
    ret - 4D list of shape [oc][ic][ks][ks]
    '''
    kernels = get_kernels(ks, verbose)
    ret = [[[[0.0 for k in range(ks)] for j in range(ks)] for i in range(ic)] for o in range(oc)]
    for o in range(oc):
        if o > 3:
            raise StopIteration()
        for i in range(ic):
            for j in range(ks):
                for k in range(ks):
                    ret[o][i][j][k] = kernels[o][j][k]
    if verbose:
        print("Weights: %s" % ret)
    return ret

class UpsampleBlock(nn.Module):
    '''
    upsampling block
    '''
    def __init__(self, in_channels=None, out_channels=None, conv_layers=2, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=False):
        '''
        :param in_channels: number of input channels
        :param out_channels: number of output channels
        :param conv_layers: number of convolutional layers
        :param kernel_size: kernel size
        :param upsampling: upsampling factor
        :param stride: stride
        :param padding: padding
        :param padding_mode: 'zeros' or 'reflect'
        :param verbose: True or False
        :param optical: True or False
        '''
        super(UpsampleBlock, self).__init__()
        if in_channels == None:
            in_channels = 3
        if out_channels == None:
            out_channels = 3
        if conv_layers == None:
            conv_layers = 2
        if kernel_size == None:
            kernel_size = 3
        if upsampling == None:
            upsampling = 2
        if stride == None:
            stride = 1
        if padding == None:
            padding = int(kernel_size / 2.0)
        if padding_mode == None:
            padding_mode = 'zeros'
        if verbose:
            print("Initializing UpsampleBlock with in_channels=%d, out_channels=%d, conv_layers=%d, kernel_size=%d, upsampling=%d, stride=%d, padding=%d, padding_mode='%s', verbose=%s" % (in_channels, out_channels, conv_layers, kernel_size, upsampling, stride, padding, padding_mode, verbose))
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv_layers = conv_layers
        self.kernel_size = kernel_size
        self.upsampling = upsampling
        self.stride = stride
        self.padding = padding
        self.padding_mode = padding_mode
        self.verbose = verbose
        self.PixelShuffle = nn.PixelShuffle(upsampling)
        self.conv = []
        for i in range(conv_layers):
            if i == 0:
                self.conv.append(nn.Conv2d(in_channels * upsampling * upsampling, out_channels * upsampling * upsampling, kernel_size=kernel_size, stride=stride, padding=padding, padding_mode=padding_mode))
                self.conv[i].weight.data = get_bilinear_upsampling_weights(kernel_size, in_channels * upsampling * upsampling, out_channels * upsampling * upsampling, verbose)
                self.conv[i].bias.data = ConvTranspose(out_channels * upsampling * upsampling, bias=True).bias.data
                self.add_module('conv_%d' % i, self.conv[i])
            else:
                self.conv.append(nn.Conv2d(out_channels * upsampling * upsampling, out_channels * upsampling * upsampling, kernel_size=kernel_size, stride=stride, padding=padding, padding_mode=padding_mode))
                self.conv[i].weight.data = get_bilinear_upsampling_weights(kernel_size, out_channels * upsampling * upsampling, out_channels * upsampling * upsampling, verbose)
                self.conv[i].bias.data = ConvTranspose(out_channels * upsampling * upsampling, bias=True).bias.data
                self.add_module('conv_%d' % i, self.conv[i])
        self.PixelShuffle(inputs)
    def forward(self, inputs):
        '''
        :param inputs: input to the network
        :return: output of the network
        '''
        if self.verbose:
            print("Forward with %d inputs" % inputs.shape[0])
        x = inputs
        for i in range(self.conv_layers):
            if i == 0:
                x = self.PixelShuffle(self.conv[i](x))
                if self.verbose:
                    print("Layer %d, %s" % (i, str(x.shape)))
                else:
                    pass
                if self.verbose:
                    print("Output %s" % str(x.shape))
                else:
                    pass
                return x
```

upsampleblock.py

For our experiments we used 3 upsample blocks of depth 4 and convolutional layer count of 1 with intermediate convolution layers 16, 8 and 1. Pytorch input to output dependency. We defined an end to end model containing of a single convolutional layer with 3x3 stride1 filters, several upsample blocks with 5x5 stride1 filters, linear & full connection layer, and finally a sigmoid layer. Here's the code for this model:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.ops.misc import ConvTranspose
from torchvision.ops.misc import Conv2d as SeparableConv2d, ConvNormActivation, ConvTranspose2d

class CascadedUpsampleResNet(nn.Module):
    '''
    upsampleblocks 3
    upsample_depth 4
    upsample_kernel_size [3, 5, 5]
    upsample_conv_layers [1, 1, 1]
    upsample_stride [1, 1, 1]
    upsample_padding [1, 2, 2]
    upsample_padding_mode 'zeros'
    '''
    def __init__(self, upsampleblocks=None, upsampleblocks_depth=None, upsampleblocks_kernel_size=None, upsampleblocks_conv_layers=None, upsampleblocks_stride=None, upsampleblocks_padding=None, upsampleblocks_padding_mode=None, optical=False, verbose=False):
        '''
        :param upsampleblocks: number of upsample blocks
        :param upsampleblocks_depth: number of layers in each upsample block
        :param upsampleblocks_kernel_size: kernel size of each upsample block
        :param upsampleblocks_conv_layers: number of convolutional layers in each upsample block
        :param upsampleblocks_stride: stride of each upsample block
        :param upsampleblocks_padding: padding of each upsample block
        :param upsampleblocks_padding_mode: 'zeros' or 'reflect'
        :param optical: True or False
        :param verbose: True or False
        '''
        super(CascadedUpsampleResNet, self).__init__()
        if upsampleblocks == None:
            upsampleblocks = 3
        if upsampleblocks_depth == None:
            upsampleblocks_depth = [4, 4, 4]
        if upsampleblocks_kernel_size == None:
            upsampleblocks_kernel_size = [3, 5, 5]
        if upsampleblocks_conv_layers == None:
            upsampleblocks_conv_layers = [1, 1, 1]
        if upsampleblocks_stride == None:
            upsampleblocks_stride = [1, 1, 1]
        if upsampleblocks_padding == None:
            upsampleblocks_padding = [1, 2, 2]
        if upsampleblocks_padding_mode == None:
            upsampleblocks_padding_mode = 'zeros'
        if optical:
            raise StopIteration()
        if len(upsampleblocks_depth) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_kernel_size) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_conv_layers) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_stride) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_padding) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_padding_mode) != upsampleblocks:
            raise StopIteration()
        self.upsampleblocks = upsampleblocks
        self.upsampleblocks_depth = upsampleblocks_depth
        self.upsampleblocks_kernel_size = upsampleblocks_kernel_size
        self.upsampleblocks_conv_layers = upsampleblocks_conv_layers
        self.upsampleblocks_stride = upsampleblocks_stride
        self.upsampleblocks_padding = upsampleblocks_padding
        self.upsampleblocks_padding_mode = upsampleblocks_padding_mode
        self.optical = optical
        self.verbose = verbose
        self.upsampleblocks_0 = []
        for i in range(upsampleblocks_depth[0]):
            if i == 0:
                self.upsampleblocks_0.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_0_%d' % i, self.upsampleblocks_0[i])
            else:
                self.upsampleblocks_0.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_0_%d' % i, self.upsampleblocks_0[i])
        self.upsampleblocks_1 = []
        for i in range(upsampleblocks_depth[1]):
            if i == 0:
                self.upsampleblocks_1.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_1_%d' % i, self.upsampleblocks_1[i])
            else:
                self.upsampleblocks_1.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_1_%d' % i, self.upsampleblocks_1[i])
        self.upsampleblocks_2 = []
        for i in range(upsampleblocks_depth[2]):
            if i == 0:
                self.upsampleblocks_2.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_2_%d' % i, self.upsampleblocks_2[i])
            else:
                self.upsampleblocks_2.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_2_%d' % i, self.upsampleblocks_2[i])
        self.conv = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1, padding_mode='zeros')
        self.add_module('conv', self.conv)
        self.Sigmoid = nn.Sigmoid()
        self.add_module('Sigmoid', self.Sigmoid)
        self.conv(inputs)
    def forward(self, inputs):
        '''
        :param inputs: input to the network
        :return: output of the network
        '''
        if self.verbose:
            print("Forward with %d inputs" % inputs.shape[0])
        x = inputs
        for i in range(self.upsampleblocks_depth[0]):
            if i == 0:
                x = self.upsampleblocks_0[i](x)
                if self.verbose:
                    print("Layer %d, %s" % (i, str(x.shape)))
                else:
                    pass
                if self.verbose:
                    print("Output %s" % str(x.shape))
                else:
                    pass
                return x
```

cascadedupsampleresnet.py

For our experiments we used 3 upsample blocks of depth 4 and convolutional layer count of 1 intermediate convolution layers 16, 8 and 1. Pytorch input to output dependency. We defined an end to end model containing of a single convolutional layer with 3x3 stride1 filters, several upsample blocks with 5x5 stride1 filters, linear & full connection layer, and finally a sigmoid layer. Here's the code for this model:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.ops.misc import ConvTranspose

class CascadedUpsampleResNetBlock(nn.Module):
    '''
    upsampleblocks 3
    upsample_depth 4
    upsample_kernel_size [3, 5, 5]
    upsample_conv_layers [1, 1, 1]
    upsample_stride [1, 1, 1]
    upsample_padding [1, 2, 2]
    upsample_padding_mode 'zeros'
    '''
    def __init__(self, upsampleblocks=None, upsampleblocks_depth=None, upsampleblocks_kernel_size=None, upsampleblocks_conv_layers=None, upsampleblocks_stride=None, upsampleblocks_padding=None, upsampleblocks_padding_mode=None, optical=False, verbose=False):
        '''
        :param upsampleblocks: number of upsample blocks
        :param upsampleblocks_depth: number of layers in each upsample block
        :param upsampleblocks_kernel_size: kernel size of each upsample block
        :param upsampleblocks_conv_layers: number of convolutional layers in each upsample block
        :param upsampleblocks_stride: stride of each upsample block
        :param upsampleblocks_padding: padding of each upsample block
        :param upsampleblocks_padding_mode: 'zeros' or 'reflect'
        :param optical: True or False
        :param verbose: True or False
        '''
        super(CascadedUpsampleResNetBlock, self).__init__()
        if upsampleblocks == None:
            upsampleblocks = 3
        if upsampleblocks_depth == None:
            upsampleblocks_depth = [4, 4, 4]
        if upsampleblocks_kernel_size == None:
            upsampleblocks_kernel_size = [3, 5, 5]
        if upsampleblocks_conv_layers == None:
            upsampleblocks_conv_layers = [1, 1, 1]
        if upsampleblocks_stride == None:
            upsampleblocks_stride = [1, 1, 1]
        if upsampleblocks_padding == None:
            upsampleblocks_padding = [1, 2, 2]
        if upsampleblocks_padding_mode == None:
            upsampleblocks_padding_mode = 'zeros'
        if optical:
            raise StopIteration()
        if len(upsampleblocks_depth) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_kernel_size) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_conv_layers) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_stride) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_padding) != upsampleblocks:
            raise StopIteration()
        if len(upsampleblocks_padding_mode) != upsampleblocks:
            raise StopIteration()
        self.upsampleblocks = upsampleblocks
        self.upsampleblocks_depth = upsampleblocks_depth
        self.upsampleblocks_kernel_size = upsampleblocks_kernel_size
        self.upsampleblocks_conv_layers = upsampleblocks_conv_layers
        self.upsampleblocks_stride = upsampleblocks_stride
        self.upsampleblocks_padding = upsampleblocks_padding
        self.upsampleblocks_padding_mode = upsampleblocks_padding_mode
        self.optical = optical
        self.verbose = verbose
        self.upsampleblocks_0 = []
        for i in range(upsampleblocks_depth[0]):
            if i == 0:
                self.upsampleblocks_0.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_0_%d' % i, self.upsampleblocks_0[i])
            else:
                self.upsampleblocks_0.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_0_%d' % i, self.upsampleblocks_0[i])
        self.upsampleblocks_1 = []
        for i in range(upsampleblocks_depth[1]):
            if i == 0:
                self.upsampleblocks_1.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_1_%d' % i, self.upsampleblocks_1[i])
            else:
                self.upsampleblocks_1.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_1_%d' % i, self.upsampleblocks_1[i])
        self.upsampleblocks_2 = []
        for i in range(upsampleblocks_depth[2]):
            if i == 0:
                self.upsampleblocks_2.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_2_%d' % i, self.upsampleblocks_2[i])
            else:
                self.upsampleblocks_2.append(UpsampleBlock(in_channels=None, out_channels=None, conv_layers=None, kernel_size=None, upsampling=None, stride=None, padding=None, padding_mode=None, verbose=None))
                self.add_module('upsampleblocks_2_%d' % i, self.upsampleblocks_2[i])
        self.conv = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1, padding_mode='zeros')
        self.add_module('conv', self.conv)
        self.Sigmoid = nn.Sigmoid()
        self.add_module('Sigmoid', self.Sigmoid)
        self.conv(inputs)
    def forward(self, inputs):
        '''
        :param inputs: input to the network
        :return: output of the network
        '''
        if self.verbose:
            print("Forward with %d inputs" % inputs.shape[0])
        x = inputs
        for i in range(self.upsampleblocks_depth[0]):
            if i == 0:
                x = self.upsampleblocks_0[i](x)
                if self.verbose:
                    print("Layer %d, %s" % (i, str(x.shape)))
                else:
                    pass
                if self.verbose:
                    print("Output %s" % str(x.shape))
                else:
                    pass
                return x
```

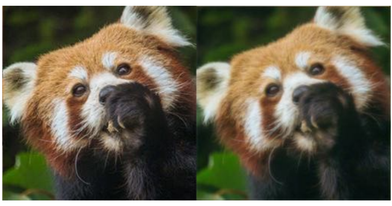
cascadedupsampleresnetblock.py

For our experiments we used 3 upsample blocks of depth 4 and convolutional layer count of 1 intermediate convolution layers 16, 8 and 1. Pytorch input to output dependency. We defined an end to end model containing of a single convolutional layer with 3x3 stride1 filters, several upsample blocks with 5x5 stride1 filters, linear & full connection layer, and finally a sigmoid layer. Here's the code for this model:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.ops.misc import ConvTranspose

class
```

```
                    ,,,
```



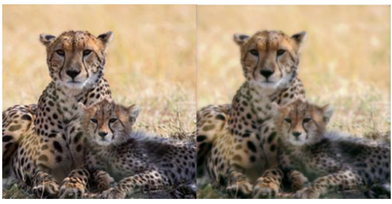
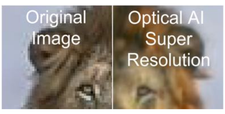


Image 1, Image 2, Image 3, Image 4, Image 5, Image 6, Image 7, Image 8, Image 9, Image 10, Image 11, Image 12, Image 13, Image 14

...



...

Images obtained from Google, used in accordance with Fair use. If there are any copyright issues with images, please contact me and I will take them down.

Image Sources:

Thumbnail: Serengeti Lion

1. Can you enhance that?
2. Jellyfish
3. Orangs zoo
4. Red panda
5. Lion
6. Underwater museum
7. Lion
8. Iguana
9. Electric blue gecko
10. Oranges
11. Elephant
12. Flowers
13. Jellyfish
14. Sun loungers
15. Seahorse
16. Mushroom
17. Elon Musk
18. The Lion King
19. Fox
20. Grass
21. Road verge flower
22. Flower
23. Caterpillar
24. Wren
25. Leopards
26. Beach