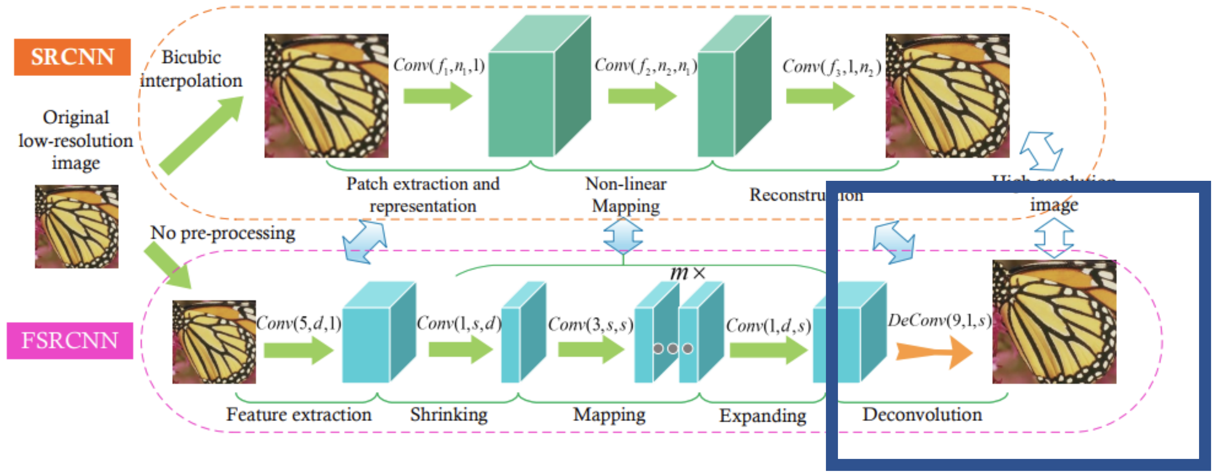
FSRCNN

Accelerating the Super-Resolution Convolutional Neural Network

3 Implementations to speed up SR task

Compared to SRCNN

1. Deconvolution



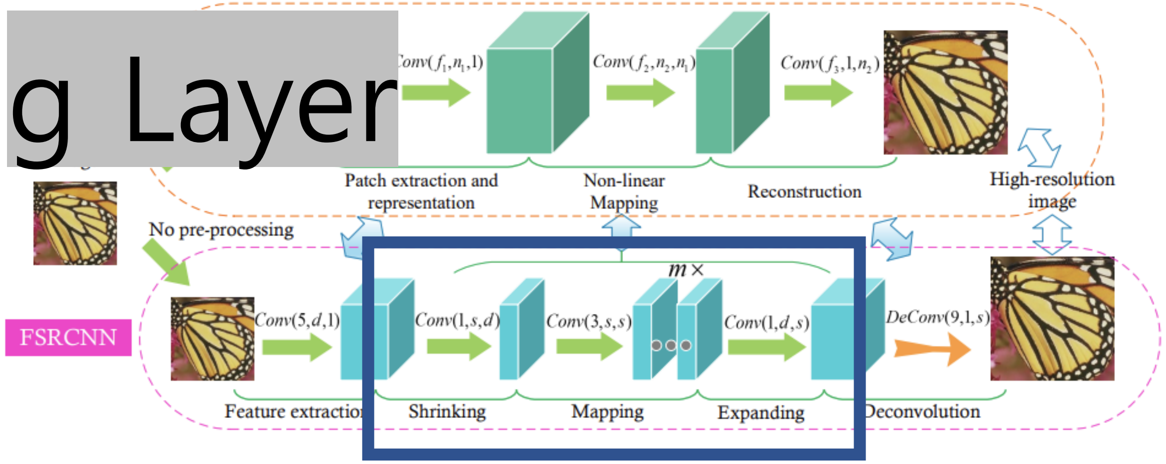
Replaces…
- Bicubic interpolation to upsample image
- The network has to process upsampled HR feature map

Therefore…
- Diverse upsampling kernels for better upsampling
- Network can work with LR feature map = less computation
- The first layer (feature extraction) kernel size reduced from 9 to 5 for the same effect

Speed up 8.7x

Performance increase 0.12dB

2. Multiple mapping Layer



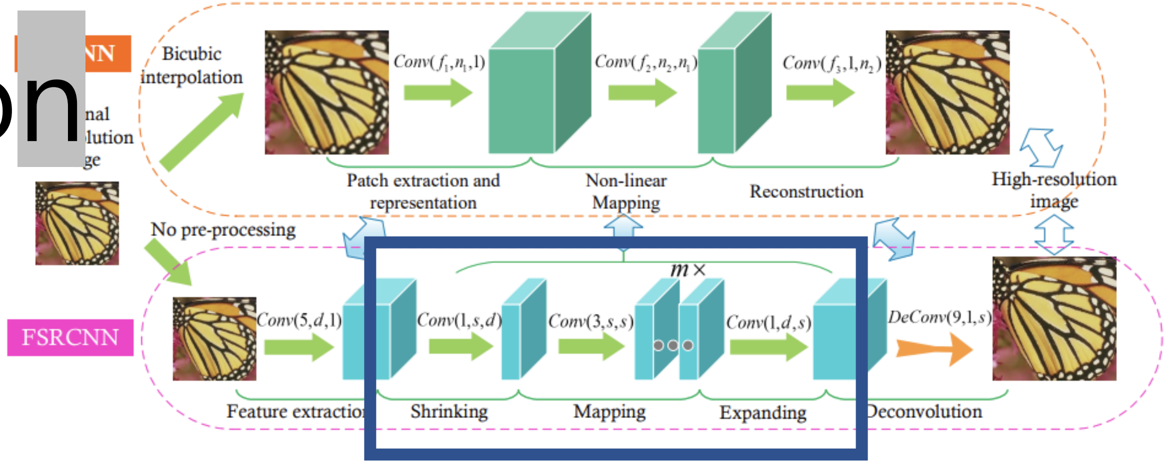
Replaces…
- 5x5 kernel Conv2d layer (And its large parameter count / computational cost)

Therefore…
- Shrinking and Expanding allows reduction of channels = reduction of parameters
- Consistent non-linear mapping with 3x3 kernel Conv2d
- Multiple mid layers to improve accuracy while also reducing network scale

Speed up 3.5x (8.7 x 3.5 = 30.1)

Performance increase 0.06dB

3. Feature Extraction



Replaces…
- 9x9 kernel Conv2d layer and its filter number

Therefore…
- 5x5 kernel Conv2d layer with less channels
- But, achieves the same effect because FSRCNN works with LR feature dimension

Speed up 1.4x (8.7 x 3.5 x 1.4 = 41.3)

Performance increase 0.05dB

Implementation: FSRCNN

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

        self.convolution = nn.Sequential(OrderedDict([
                                                     ('feature_extraction', nn.Conv2d(1, 56, kernel_size=5, stride=1, padding=2)),
                                                     ('feature_extraction_prelu', nn.PReLU()),
                                                     ('shrinking', nn.Conv2d(56, 12, kernel_size=1, stride=1, padding=0)),
                                                     ('shrinking_prelu', nn.PReLU()),
                                                     ('mapping_1', nn.Conv2d(12, 12, kernel_size=3, stride=1, padding=1)),
                                                     ('mapping_1_prelu', nn.PReLU()),
                                                     ('mapping_2', nn.Conv2d(12, 12, kernel_size=3, stride=1, padding=1)),
                                                     ('mapping_2_prelu', nn.PReLU()),
                                                     ('mapping_3', nn.Conv2d(12, 12, kernel_size=3, stride=1, padding=1)),
                                                     ('mapping_3_prelu', nn.PReLU()),
                                                     ('mapping_4', nn.Conv2d(12, 12, kernel_size=3, stride=1, padding=1)),
                                                     ('mapping_4_prelu', nn.PReLU()),
                                                     ('expanding', nn.Conv2d(12, 56, kernel_size=1, stride=1, padding=0)),
                                                     ('expanding_prelu', nn.PReLU())
        ]))
```

```python
        self.deconvolution = nn.ConvTranspose2d(56, 1, kernel_size=9, stride=3, padding=3)

    def _init_weight(self):
         nn.init.kaiming_normal_(self.convolution.feature_extraction.weight.data)
         nn.init.kaiming_normal_(self.convolution.shrinking.weight.data)
         nn.init.kaiming_normal_(self.convolution.mapping_1.weight.data)
         nn.init.kaiming_normal_(self.convolution.mapping_2.weight.data)
         nn.init.kaiming_normal_(self.convolution.mapping_3.weight.data)
         nn.init.kaiming_normal_(self.convolution.mapping_4.weight.data)
         nn.init.kaiming_normal_(self.convolution.expanding.weight.data)
         nn.init.kaiming_normal_(self.deconvolution.weight.data)

    def forward(self, x):
        x = self.convolution(x)

        x = self.deconvolution(x)

        return x
```

```python
        self.deconvolution = nn.ConvTranspose2d(56, 1, kernel_size=9, stride=3, padding=3)

    def _init_weight(self):
        nn.init.kaiming_normal_(self.convolution.feature_extraction.weight.data)
        nn.init.kaiming_normal_(self.convolution.shrinking.weight.data)
        nn.init.kaiming_normal_(self.convolution.mapping_1.weight.data)
        nn.init.kaiming_normal_(self.convolution.mapping_2.weight.data)
        nn.init.kaiming_normal_(self.convolution.mapping_3.weight.data)
        nn.init.kaiming_normal_(self.convolution.mapping_4.weight.data)
        nn.init.kaiming_normal_(self.convolution.expanding.weight.data)
        nn.init.kaiming_normal_(self.deconvolution.weight.data)

    def forward(self, x):
        x = self.convolution(x)

        x = self.deconvolution(x)

        return x
```

Implementation: Training

```python
def create_training_dataset(image_path, images):

    sub_img_dim = 48
    scales = [1, 0.9, 0.8, 0.7, 0.6, 0.2]
    rotations = [0, 90, 180, 270]

    training_image_path = image_path[:-1] + '_training3/'

    for image in images:
        img = Image.open(image_path + image)
        img_width, img_height = img.size
        img_name, img_type = image.split('.')

        for s in scales:
            resized_img = img.resize((int(img_width*s), int(img_height*s)))
            for r in rotations:
                rotated_img = resized_img.rotate(r, expand=1)
                rotated_img_width, rotated_img_height = rotated_img.size
                for sub_x in range(0, rotated_img_width-sub_img_dim, sub_img_dim):
                    for sub_y in range(0, rotated_img_height-sub_img_dim, sub_img_dim):
                        cropped_img = rotated_img.crop((sub_x, sub_y, sub_x+sub_img_dim, sub_y+sub_img_dim))
                        cropped_img.save(f'{training_image_path}{img_name}_{s}_{r}_{sub_x}_{sub_y}.{img_type}')
```

1. Note that the dataset is cropped original images

although the paper has dimensions 7^2 and 19^2 for scale factor=3, for simplicity the current implantation is done with 16^2 and 48^2

```python
class T91_Dataset(Dataset):
    def __init__(self,
                 image_path,
                 images):

        self.image_path = image_path
        self.images = images
        self.transforms = transforms.ToTensor()

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

    def __getitem__(self, index):
        HR_image = Image.open(
            os.path.join(self.image_path, self.images[index])
        ).convert("YCbCr")

        HR_image, _, _ = HR_image.split()

        LR_image = HR_image.resize((16,16))

        LR_image = self.transforms(LR_image)
        HR_image = self.transforms(HR_image)

        return LR_image, HR_image
```

2. Create dataset of both high-resolution and low-resolution image

low resolution image is downscaled high resolution image

image called in as YCbCr because we extract just 1 channel

```python
class General_Dataset(Dataset): # T91 + General-100
    def __init__(self,
                 image_path1,
                 image_path2,
                 images1,
                 images2):

        self.images = [image_path1+'/'+image for image in images1] + [image_path2+'/'+image for image in images2]
        self.images_len = len(self.images)
        self.transforms = transforms.ToTensor()

    def __len__(self):
        return self.images_len

    def __getitem__(self, index):
        HR_image = Image.open(
            os.path.join(self.images[index])
        ).convert("YCbCr")

        HR_image, _, _ = HR_image.split()

        LR_image = HR_image.resize((16,16))

        LR_image = self.transforms(LR_image)
        HR_image = self.transforms(HR_image)

        return LR_image, HR_image
```

3. For fine-tuning with General-100 images as well, general_dataset is made.

a little effort for a dataset that can index both T91 and General-100

4. with the help of link in stackoverflow, created random split into train/valid set

```
#https://stackoverflow.com/questions/50544730/how-do-i-split-a-custom-dataset-into-training-and-test-datasets

indices = list(range(t91_dataset.__len__()))
split = int(np.floor(0.2 * t91_dataset.__len__()))
np.random.seed(2021)
np.random.shuffle(indices)
train_indices , valid_indices = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_indices)
valid_sampler = SubsetRandomSampler(valid_indices)

train_loader = DataLoader(t91_dataset, batch_size=128, sampler=train_sampler)
valid_loader = DataLoader(t91_dataset, batch_size=128, sampler=valid_sampler)
```

5. layer-wise learning rate and optimizer changed to AdamW

```
loss_fn = nn.MSELoss()
optimizer = torch.optim.AdamW([{'params':fsrcnn.convolution.parameters()},
                               {'params':fsrcnn.deconvolution.parameters(),'lr':1e-4}], lr=1e-3)
```

6. learning rate halved for fine-tuning with general_dataset

```
loss_fn = nn.MSELoss()
optimizer = torch.optim.AdamW([{'params':fsrcnn.convolution.parameters()},
                               {'params':fsrcnn.deconvolution.parameters(),'lr':5e-5}], lr=5e-4)

epochs=20
```

```python
fsrcnn.eval()

lr_image = Image.open(
    os.path.join('./Set14','LRbicx3', 'barbara.png')
    ).convert("YCbCr")
hr_image = Image.open(
    os.path.join('./Set14','original', 'barbara.png')
    ).convert("YCbCr")
bicubic = lr_image.resize((720, 576), resample=PIL.Image.BICUBIC)
lr_image, _, _ = lr_image.split()
hr_image, _, _ = hr_image.split()

transform = transforms.ToTensor()

lr_image = transform(lr_image)
hr_image = transform(hr_image)

lr_image = lr_image.to(device)
hr_image = hr_image.to(device)

upsample = fsrcnn(lr_image.unsqueeze(0))
loss_fn = torch.nn.MSELoss()
mse = loss_fn(upsample, hr_image)
psnr = calc_psnr(upsample, hr_image)
```

7. Test prediction with Set 14 images

bicubic interpolation upscaled image is used to support the output of the model upsampled image (why?)

```python
plt.figure()

f, axarr = plt.subplots(1,3)
f.set_figheight(30)
f.set_figwidth(30)

hr_image = Image.open(
    os.path.join('./Set14','original', 'barbara.png')
    ).convert("RGB")

bicubic = transform(bicubic).to(device)
upsample = torch.cat((upsample.squeeze(0), bicubic[1,:, :].unsqueeze(0), bicubic[2,:,:].unsqueeze(0)),0)
print(upsample.shape)
bicubic = bicubic.cpu().numpy()
upsample = upsample.detach().cpu().numpy()

axarr[0].imshow(hr_image)
axarr[1].imshow(ycbcr2rgb(255*bicubic.transpose(1,2,0)))
axarr[2].imshow(ycbcr2rgb(255*upsample.transpose(1,2,0)))

print(f"mse:{mse}")
print(f"psnr:{psnr}")
```

8. For visualization.

needed to incorporate bicubic and upsample image

then convert to RGB

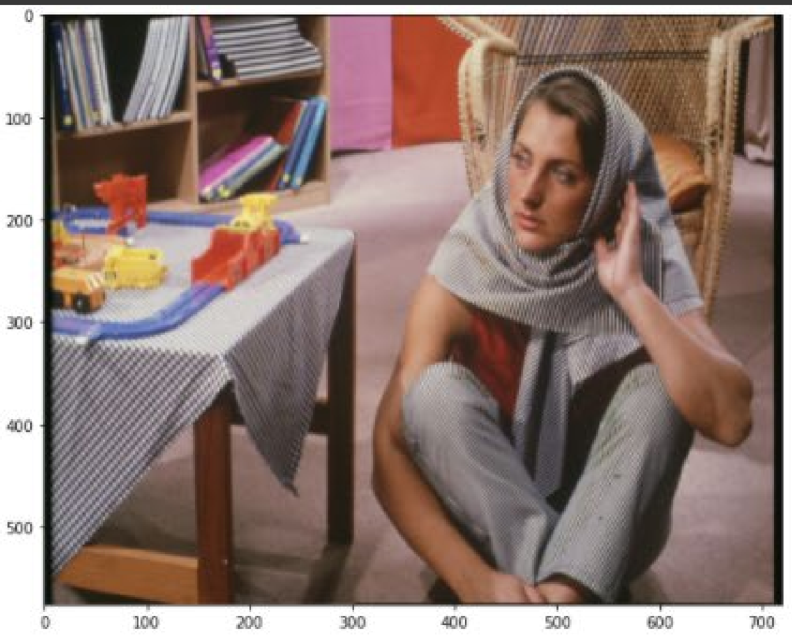
0. with the help of these references

```python
#https://github.com/yjn870/FSRCNN-pytorch/blob/master/

def preprocess(img, device):
    img = np.array(img).astype(np.float32)
    ycbcr = convert_rgb_to_ycbcr(img)
    x = ycbcr[..., 0]
    x /= 255.
    x = torch.from_numpy(x).to(device)
    x = x.unsqueeze(0).unsqueeze(0)
    return x, ycbcr
def calc_psnr(img1, img2):
    return 10. * torch.log10(1. / torch.mean((img1 - img2) ** 2))

#https://stackoverflow.com/questions/34913005/color-space-mapping-ycbcr-to-rgb
def ycbcr2rgb(im):
    xform = np.array([[1, 0, 1.402], [1, -0.34414, -.71414], [1, 1.772, 0]])
    rgb = im.astype(np.float)
    rgb[:,:,[1,2]] -= 128
    rgb = rgb.dot(xform.T)
    np.putmask(rgb, rgb > 255, 255)
    np.putmask(rgb, rgb < 0, 0)
    return np.uint8(rgb)
```

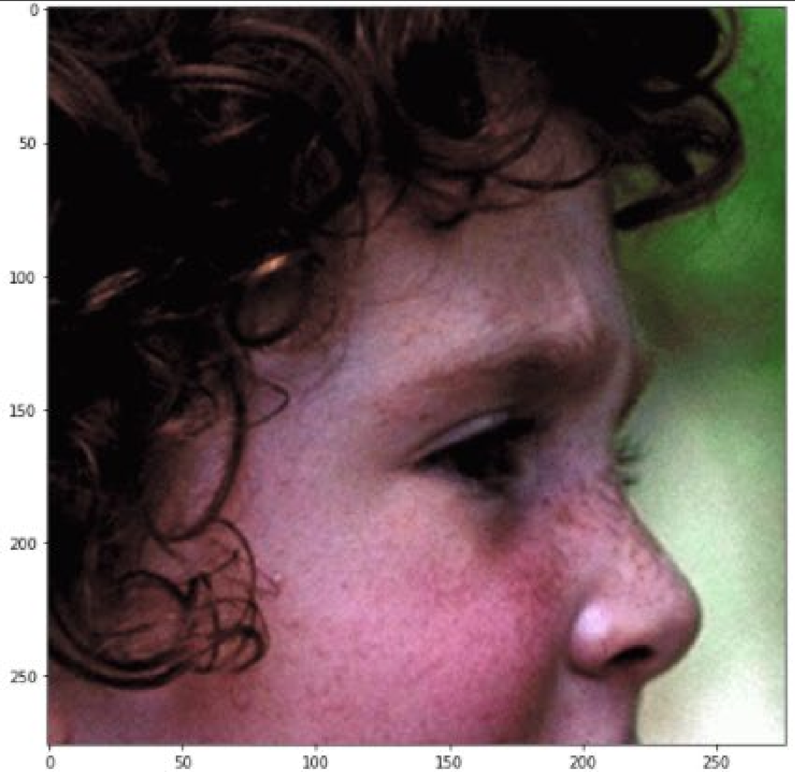
Implementation: Results



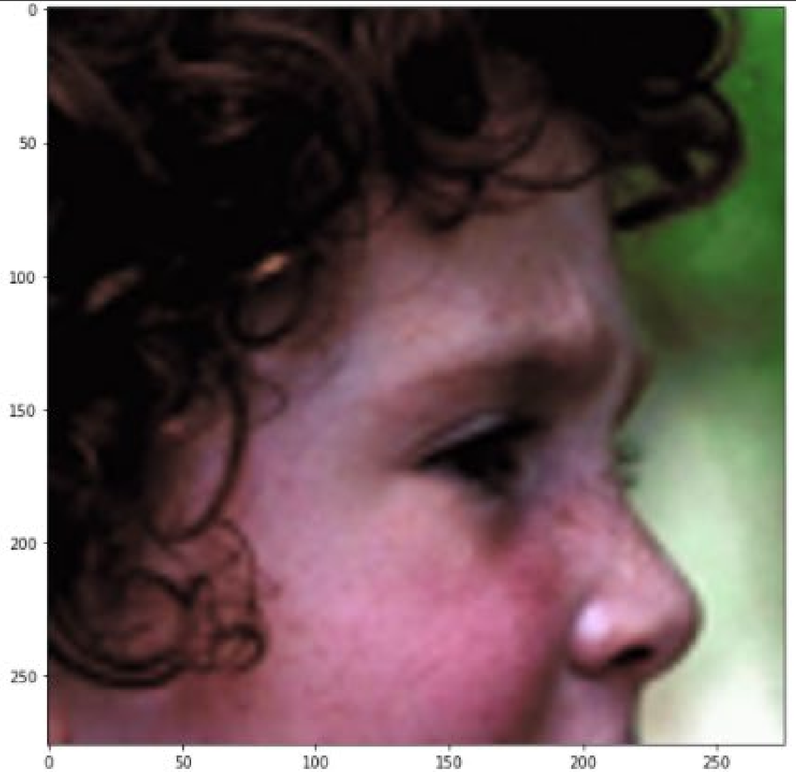
Original

Bicubic

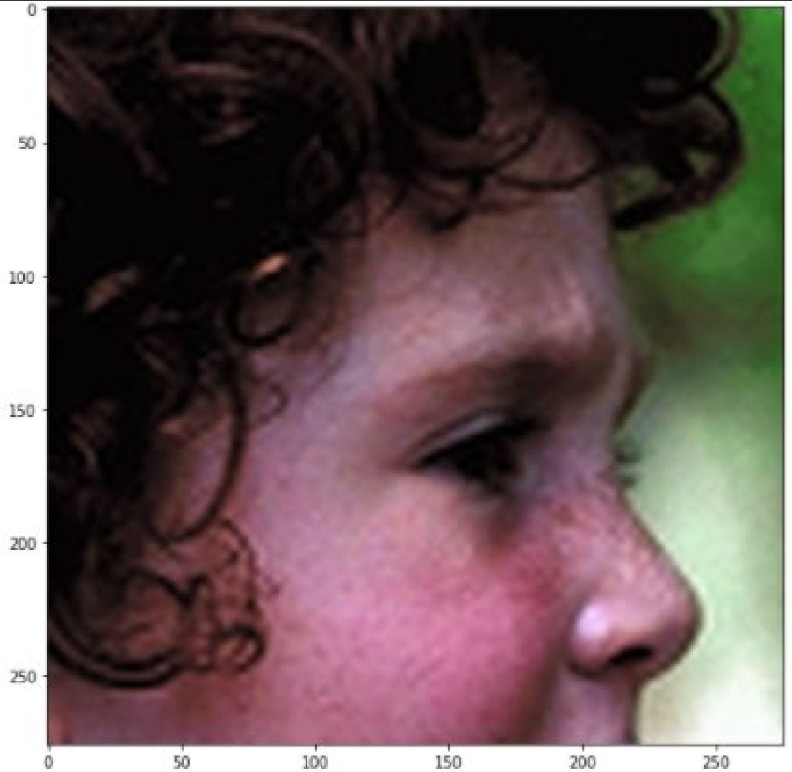
FSRCNN



Original

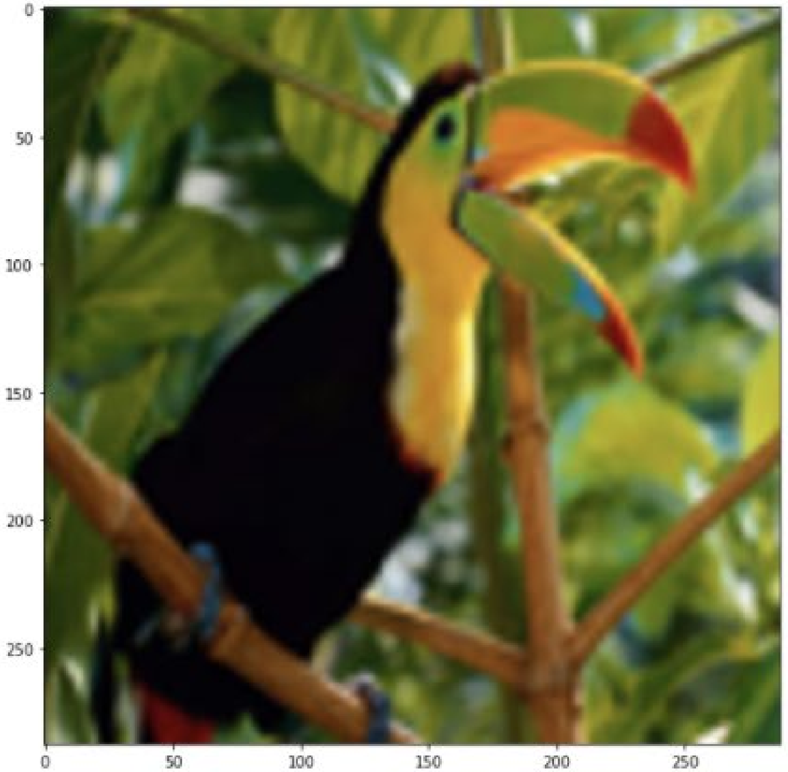


Bicubic



FSRCNN

Original



Bicubic

FSRCNN