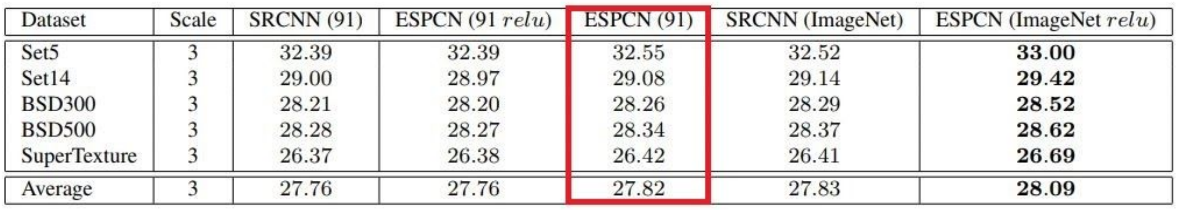
Reproducing “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network”

Luuk Balkenende
L.W.P.Balkenende@student.tudelft.nl
4375157

Sieger Falkena
S.T.Falkena@student.tudelft.nl
4293681

Luc Kloosterman
L.J.A.Kloosterman@student.tudelft.nl
4697006

This article describes a replication of Table 1 from the [Real-time Single Image Super-Resolution] [1] paper trained on the [yang91] dataset with a validation on different [datasets]. The table below shows the original table from the paper and highlights the results which are attempted to reproduce.



Dataset	Scale	SRCNN (91)	ESPCN (91 $relu$)	ESPCN (91)	SRCNN (ImageNet)	ESPCN (ImageNet $relu$)
Set5	3	32.39	32.39	32.55	32.52	**33.00**
Set14	3	29.00	28.97	29.08	29.14	**29.42**
BSD300	3	28.21	28.20	28.26	28.29	**28.52**
BSD500	3	28.28	28.27	28.34	28.37	**28.62**
SuperTexture	3	26.37	26.38	26.42	26.41	**26.69**
Average	3	27.76	27.76	27.82	27.83	**28.09**

The reproduction is done using a Pytorch implementation which is written from scratch. Only the information given in the paper is used. Some hyperparameters are tuned as those are not mentioned in the paper. The code is developed on Google Colab and is compatible to run with a GPU.

Problem definition of the paper

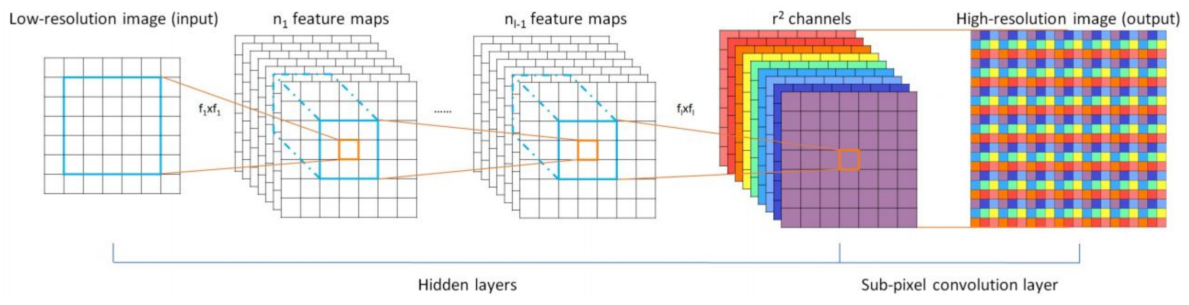
Several models which are capable of upscaling single images already exist. [2] [3] [4] However, in these methods the super-resolution (SR) operation is performed in high resolution (HR) space. According to the writers of the paper: '*this is sub-optimal and adds computation complexity*'. The goal is to find a more efficient method to upscale images. More specifically, the goal is to make this process fast enough so that it can be applied to real-time video material.

Experiment setup as proposed by the authors

The authors of the paper propose a CNN architecture where the feature maps are extracted in low resolution (LR) space. Only at the very end of the network the resolution will be increased. The advantage of this network is two fold:

- The computational complexity of the whole model is low.
- Better and more complex SR operations are learned compared to other models.

The proposed CNN architecture, called ESPCN, is shown below. As can be seen, the ESPCN consists of three convolutional layers, two normal convolutional layers and one sub-pixel convolutional layer which aggregates the feature maps from LR space and performs the SR operation. This last layer is called the sub-pixel convolution layer. In the first layer the Y channel of a YCrCb LR image is taken as the input and is convolved with a kernel size of 5x5 to 64 different output channels. After this, two convolutional layers are used with a kernel size of 3x3 which reduces the number of channels to the square of the scaling factor. The last layer shuffles the pixels to obtain the final SR image.



Implementation of the code

The actual code is implemented from scratch in this [notebook], where only the information given in the paper is used. In this section, the steps which are taken in order to reproduce the desired results of the paper together with the python code are explained.

Running the experiment on Google Colab

The notebook is running remotely on the Google Colab platform. Therefore, to save and access the trained model, we needed to mount the Google drive. We used the following code snippet to set up a local drive on our computer. Furthermore, the packages needed to run the whole notebook are imported.

```
from torchvision import transforms
from google.colab import drive
from torch.utils import data
from PIL import Image

import matplotlib.pyplot as plt
import PIL.Image as pil_image
import torch.nn as nn
import torchvision
import numpy as np
import torch
import time
import cv2
import os

drive.mount('/content/gdrive')
path ='/content/gdrive/My Drive/deep_learning_group_7/Final'
os.chdir(path)
```

ESPCN Architecture

Below, the ESPCN architecture as defined in the paper can be found. For the activation function a `tanh` is used, as the authors indicated this will lead to better results. The final sub-pixel convolution layer is divided into a normal convolutional layer (`conv3`) and a layer which performs the SR operation (`upsample`). Next to defining the model this cell checks the availability of a GPU and stores the model on the GPU if it is available. Note that for the implementation, an upscaling factor of 3 is used.

```
# Parameters
r = 3 #scaling factor

class SuperResConvNet(nn.Module):
  'ESPCN network architecture'
  def __init__(self):
    'Initialize layers'
    super(SuperResConvNet, self).__init__()
    self.conv1 = nn.Conv2d(1,64, kernel_size=5, stride=1, padding=2)
    self.conv2 = nn.Conv2d(64,32, kernel_size=3, stride=1, padding=1)
    self.conv3 = nn.Conv2d(32,1*r**2, kernel_size=3, stride=1, padding=1)
    self.upsample = nn.PixelShuffle(r)

  def forward(self, y):
    'Define forward pass'
    y = torch.tanh(self.conv1(y))
    y = torch.tanh(self.conv2(y))
    y = self.conv3(y)
    y = self.upsample(y)
    return y

# Check for GPU availability
if torch.cuda.is_available():
  print("Using GPU")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define model and send to device
ConvNet = SuperResConvNet()
ConvNet.to(device)
```

Datasets and data processing

In order to train the above defined model, the images in the [yang91] dataset are downsampled by a factor of 3 and saved on the drive. More about this operation can be found in the next section.

The model will not train on the whole images, but on small parts of the images, as explained in the paper. We refer to these small parts as patches in this reproduction. In order to reduce memory issues, it has been chosen to only save a list of names of the available patches per image, instead of the individual patches themselves. As can be seen in the code snippet below, the function `getPatchList` is searching for the available patches per image, while `PatchList_get` is saving the information of the former function into a list. During training time, the function `createPatch` is used to transform the entries of the list of available patches into real image patches. Finally, the bicubic upsampling function is used to load low resolution images and upsample them to high resolution images. Later, during visualisation, the luminance channel is substituted with the one upsampled by the model, while the Cr and Cb channels are kept. This makes that the images only trained on luminance can be displayed in color. Noticeably, multiple functions need to be called on the sets of images from different directories. For simplicity, all of these functions are accommodated in one class.

```
# Define directories
slide_subfolders = ['yang91/T91/', #trainingset
                    'x3.0/Set5/', 'x3.0/Set14/', # Testsets
                    'x3.0/BSD300/', 'x3.0/BSD500/', 'x3.0/SuperText136/']

#width and height of patches
x = 17

class Slide:
  'Combines functions for loading and preparing images for training and testing'
  def __init__(self, path, slide_subfolders, count):
    self.count = count
    self.dir = path + '/CVPR2016_ESPCN_OurBenchMarkResult/Ours/' + slide_subfolders

    self.namelist = self.deleteLRImage()
    if count == 0:
        self.patchlist = self.patchlist_get()

  def removePng(f):
    'returns filename without png'
    filename_parts = f[:-4]
    return filename_parts

  def getList(self):
    'returns list of filenames'
    return [Slide.removePng(f) for f in os.listdir(self.dir) if f.endswith(".png")]

  def deleteLRImage(self):
    'deletes LR images from list if they are already made (which they are)'
    name_list_2 = Slide.getList(self)
    name_list_1 = [x for x in name_list_2 if "lr" not in x ]
    name_list =  [x for x in name_list_1 if "lowRes" not in x ]
    return name_list

  def getPatchList(self, img_name):
    'returns patch list of one image'
    #get filenames and load images
    filename = self.dir + img_name
    lr_img = cv2.cvtColor(cv2.imread(filename+'_lr.png'), cv2.COLOR_BGR2RGB)

    #parameters
    stride_lr = x-np.sum((5%2,3%2,3%2))
    tot_img_d = int(lr_img.shape[0]/stride_lr),
int(lr_img.shape[1]/stride_lr)    #amount of image in height and width respectively
    tot_img = tot_img_d[0]*tot_img_d[1]     #total amount of images

    #create list for current image
    patch_list = []
    for i in range(tot_img_d[0]-1):
        for j in range(tot_img_d[1]-1):
            patch_list.append([img_name, i,j])
    return patch_list

  def patchlist_get(self):
    'create patch_list'
    patch_list = []
    for i in range(len(self.namelist)):
        patch_list.extend(self.getPatchList(self.namelist[i]))
    print('Found', len(patch_list), 'trainable patches out of', len(self.namelist),
'images.')
    return patch_list

  def createPatch(self, name):
    'returns a patch'
    img_name = name[0]
    patch_name = name[1], name[2]

    #get corresponding images
    filename = self.dir + img_name
    hr_img = cv2.cvtColor(cv2.imread(filename+'.png'), cv2.COLOR_BGR2YCrCb)[:,:,0] #
only get Y channel from YCrCb
    lr_img = cv2.cvtColor(cv2.imread(filename+'_lr.png'), cv2.COLOR_BGR2YCR_CB)[:,:,0]

    #create hr patch
    stride_hr = (x-np.sum((5%2,3%2,3%2)))*r
    hr_patch =
hr_img[stride_hr*patch_name[0]:(stride_hr*patch_name[0]+17*r),stride_hr*patch_name[1]:(s
tride_hr*patch_name[1]+17*r)]

    #create lr patch
    stride_lr = x-np.sum((5%2,3%2,3%2))
    lr_patch =
lr_img[stride_lr*patch_name[0]:(stride_lr*patch_name[0]+17),stride_lr*patch_name[1]:(str
ide_lr*patch_name[1]+17)]

    return lr_patch, hr_patch

  def bicubic_upsampling(self, img_n):
    'Loading high and low resolution image and do a bicubic upsampling of the low
resolution image'
    #get corresponding images
    filename = self.dir + img_n
    hr_img = Image.open(filename + '.png').convert('YCbCr')
    lr_img = Image.open(filename[:-9] + '-lowRes.png').convert('YCbCr')

    #upsample
    bicubic = lr_img.resize(hr_img.size, Image.BICUBIC)

    return lr_img, bicubic, hr_img # return as pil images

def getSlideList(slide_subfolders, path):
  'Load Slide class for different directories'
  slides = []
  for i, slide in enumerate(slide_subfolders):
    slides.append(Slide(path, slide, i))
  return slides

slides = getSlideList(slide_subfolders, path)
```

PSNR calculation

PSNR is the performance metric to compare the real and the predicted high resolution images. The two functions below are used to calculate the mean squared error and the PSNR between two images. The PSNR calculation is already included here to make the code able to calculate the PSNR on validation images in between epochs during training.

```
def psnr_from_mselist(mse_list):
  'Calculate PSNR'
  mse = np.mean(mse_list)
  if mse == 0:
    return float('inf')
  else:
    return 20*np.log10(255/np.sqrt(mse))

def calc_mse(img_pred, img_hr):
  'Calculate MSE'
  return np.mean((img_pred*255 - img_hr*255)**2)
```

Dataset - Low Resolution image

In the paper, the writers are mentioning their way of downsampling the images: *'To synthesize the low-resolution samples, we blur the high-resolution images using a Gaussian filter and sub-sample them by the upscaling factor.' [1]* However, there is no mention of the size of the Gaussian blur. In this reproduction, there is chosen to set this kernel size of this blur to (5,5) as can be seen in the code below.

```python
## ONLY RUN THIS CELL IF LOW RESOLUTION IMAGES ARE NOT PRESENT IN THE DIRECTORY
def createLowRes(img_name, dir_91):
  'saves low resolution image'
  # Call HR image
  filename = dir_91 + img_name + '.png'
  hr_img = cv2.cvtColor(cv2.imread(filename), cv2.COLOR_BGR2RGB)

  # Blur HR image
  blur_img = cv2.GaussianBlur(hr_img,(5,5),0)

  # Apply subsampling
  lr_img = blur_img[::r,::r]

  # Save lr_img
  im = Image.fromarray(lr_img)
  im.save(dir_91 + img_name + '_lr.png')

for i in range(len(slides[0].namelist)):
  dir_91 = path + '/CVPR2016_ESPCN_OurBenchMarkResult/Ours/' + slide_subfolders[0]
  createLowRes(slides[0].namelist[i], dir_91)
```

Training

Parameters

The code below defines the settings for the training of the ESPCN. As described in the paper the mean squared error criterion and the Adam optimizer will be used. Furthermore, multiple training parameters and the scheduler for dynamic learning rate reduction are defined. An initial learning rate of 0.01 is used and a final learning rate of 1e-4 as described by the paper. The choice of other hyperparameters will be justified later.

```
# Loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(ConvNet.parameters(), lr=0.01)

# Parameters
num_epoch = 5000        #Amount of epochs
batch_size = 16         #Batch size
train_val_ratio = 0.95  #Training validation ratio

# Scheduler for dynamic reduction of the learning rate
threshold_mu = 1e-6     # Threshold for decreasing learning rate.
factor_value = 0.8     # Amount of decay per step, new lr = factor_value*lr.
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                       mode='min',
                                                       factor=factor_value,
                                                       patience=2,
                                                       threshold=threshold_mu,
                                                       min_lr=0.0001,
                                                       eps=1e-08,
                                                       verbose=True)
```

Data loader

In order to use the yang91 dataset during training, it is loaded into a Dataset class. This class divides the list of available patches into different batches every epoch. Furthermore the returned patches, which as told before are created by the function `createPatch`, are transformed into Torch tensors.

The Dataset class is divided into two parts. One part contains 95% of the available training patches, while the second part contains the remaining 5%. The two parts are both loaded into a Dataloader. The Dataloader which has 95% of the training patches is called `training_generator` and is used to provide the model with patches during training of the model. The other Dataloader is called `validation_generator` and is called to validate the model during training time.

```
class DataGenerator(data.Dataset):
  'Generates the dataset that is used for training the ESPCN'
  def __init__(self, slides):
    self.slides = slides
    self.transform = torchvision.transforms.Compose([
         torchvision.transforms.ToTensor()])

  def __len__(self):
    'Returns the amount of patches'
    return len(self.slides.patchlist)

  def __getitem__(self, idx):
    'Returns low and high resolution patches in the form of tensors'
    lr_patch, hr_patch = self.slides.createPatch(self.slides.patchlist[idx])

    # transform images to pytorch tensors
    lr_patch = self.transform(lr_patch)
    hr_patch = self.transform(hr_patch)

    return lr_patch, hr_patch

# Put DataGenerator in DataLoader
full_dataset = DataGenerator(slides[0])

# Split between training and validation set
train_size = int(train_val_ratio * len(full_dataset))
validation_size = len(full_dataset) - train_size
training_set, validation_set = torch.utils.data.random_split(full_dataset, [train_size,
validation_size])

# Create training and validation data loaders
training_generator      = data.DataLoader(training_set, batch_size=batch_size,
num_workers=batch_size, shuffle='True')
validation_generator    = data.DataLoader(validation_set, batch_size=1, num_workers=1,
shuffle='False')
```

Training the model

The actual training of the model is done in the code below. Every epoch consists of two parts. In the first part *(training*), the model is trained and in the second part (*validation*) the model is validated. The algorithm will repeat these two parts until it has done the number of epochs defined above.
During *training* low resolution patches are sent in batches to the GPU. On the GPU, the forward pass of the ESPCN network is executed. Next, the corresponding high resolution patches and the outputs after the forward pass are used to calculate the loss. During the backpropagation, the optimizer is used to find the gradient for every parameter. The parameters are updated correspondingly to that gradient. At last, the average loss of one epoch is stored in a list.
During *validation* the same procedure as in *training* is followed, with the exception of not executing the backpropagation. Furthermore, the PSNR of the validation patches is calculated and stored.
After the training, it will compute the training time.

```
# Initializing lists used for saving losses and PSNR
loss_list = []
```

```python
epoch_loss_list = []
val_loss_list = []
validation_psnr = []

# Start timer
t0 = time.time()

# Training loop
for epoch in range(num_epoch):
  # Switch to training mode
  ConvNet.train()
  for i, (lr_patch, hr_patch) in enumerate(training_generator):

    # Transfer training data to active device
    lr_patch, hr_patch = lr_patch.to(device), hr_patch.to(device)

    # Run the forward pass
    outputs = ConvNet(lr_patch)
    loss = criterion(outputs, hr_patch)
    loss_list.append(loss.item())

    # Backprop and perform Adam optimisation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  # Save loss every epoch
  epoch_loss = np.sum(loss_list)/len(training_generator)
  epoch_loss_list.append(epoch_loss)

  # Print epoch loss every 50 epochs
  if epoch % 50 == 0:
    print("Epoch", epoch, "loss: {}".format(epoch_loss))

  # Save model every 1000 epochs (in case Google Colab stops runtime)
  # if epoch % 1000 == 999:
  #   model_name = 'final_' + str(epoch+1) + '_epochs'
  #   path_model = path + '/saved_models/' + model_name
  #   torch.save(ConvNet.state_dict(), path_model)
  #   print('Model saved as: ', model_name)

  # Step to next step of lr-scheduler
  scheduler.step(epoch_loss)
  loss_list = []

  # Enter validation mode
  ConvNet.eval()

  # Keep track of mse for every patch, to collectively calculate PSNR per epoch
  epoch_mse = []

  with torch.no_grad():
    for i, (lr_patch, hr_patch) in enumerate(validation_generator):
      # Transfer training data to active device (GPU)
      lr_patch, hr_patch = lr_patch.to(device), hr_patch.to(device)

      # Predict output
      img_pred = ConvNet(lr_patch)

      # Calculate validation loss
      loss = criterion(img_pred, hr_patch)
      loss_list.append(loss.item())

      # Calculate mse for every sample
      img_pred = img_pred[0].cpu().numpy()
      hr_patch = hr_patch[0].cpu().numpy()
      epoch_mse.append(calc_mse(img_pred, hr_patch))

    # Calculate validation psnr on the complete epoch from all individual MSE's
    val_psnr = psnr_from_mselist(np.array(epoch_mse))
    validation_psnr.append(val_psnr)

  # Save validation loss every epoch
  val_loss = np.sum(loss_list)/len(validation_generator)
  val_loss_list.append(val_loss)

  loss_list = []

print('Training took {} seconds'.format(time.time() - t0))
print('Seconds per epoch:',(time.time()-t0)/num_epoch)
```

Plotting of training results

The script below will plot the training and validation loss for each epoch and a separate plot for the validation PSNR per epoch.

```
# Show training and validation loss of current model in memory
plt.figure(figsize=(8,8))
plt.subplot(2,1,1)
plt.title('Loss per epoch')
plt.plot(np.arange(num_epoch), epoch_loss_list, label='Training loss')
plt.plot(np.arange(num_epoch), val_loss_list, label='Validation loss')
plt.yscale("log")
plt.legend()

plt.subplot(2,1,2)
plt.title('PSNR for validation data')
plt.plot(np.arange(num_epoch), validation_psnr)

plt.show()
```

Testing

Loading the trained model

The code below is used to load the parameters of previous trained models from the ‘saved_models’ directory.

```
# Load weight in earlier defined model
model_name = 'final_test_5000_epochsweights'
path_model = path + '/saved_models/' + model_name
ConvNet.load_state_dict(torch.load(path_model))
```

Test image plot function

The function below is used to compare the predictions with the low and high resolution images. From left to right it shows the low resolution, the output of the neural network and the high resolution image respectively.

```
def generate_figure(lr_img, sr_img, hr_img):
  'Show the low resolution, upscaled and high resolution image'
  f = plt.figure(figsize=(8*3,8))
  f.add_subplot(1, 3, 1)
  plt.imshow(transforms.ToPILImage('YCbCr')(lr_img[0]).convert('RGB'))
  f.add_subplot(1, 3, 2)
  plt.imshow(transforms.ToPILImage('YCbCr')(sr_img[0]).convert('RGB'))
  f.add_subplot(1, 3, 3)
  plt.imshow(transforms.ToPILImage('YCbCr')(hr_img[0]).convert('RGB'))
```

Data loader

The Dataloader in the next code snippet is the same as the used Dataloader for training. However, it deviates from the training Dataloader in two ways. Firstly, it loads images, instead of patches (during training). Secondly, it also loads bicubic upsampled low resolution images. In the next section, it will be explained why the bicubic upsampled images are needed.

```
class DataGenerator_test(data.Dataset):
  'Generates the dataset used for testing'

  def __init__(self, slides):
    self.slides = slides
    self.transform =
torchvision.transforms.Compose([torchvision.transforms.ToTensor()])

  def __len__(self):
    'Returns the amount of test images'
    return len(self.slides.namelist)

  def __getitem__(self, idx):
    'Returns low, upsampled and high resolution testing images in the form
of tensors'
    lr_img, bicubic, hr_img =
self.slides.bicubic_upsampling(self.slides.namelist[idx])

    lr_img = self.transform(lr_img)
    hr_img = self.transform(hr_img)
    bicubic = self.transform(bicubic)

    return lr_img, bicubic, hr_img

## Put DataGenerator in DataLoader
def DataGenerator(slides):
  'Create dataloader for every test directory'
  test_set = DataGenerator_test(slides)
  test_generator = data.DataLoader(test_set, batch_size=1, shuffle=False)
  return test_generator
```

Testing

Finally, the ESPCN can be tested on the different test data sets. In the code below Set5, Set14, BSD300, BSD500 and the Supertexture136 dataset are used to reproduce the results from the paper.

The low resolution images are fed to the ESPCN, which outputs only the predicted high resolution Y (luminance) channel of the input images. Next, the Y channels of the bicubic upsampled images are replaced by these predicted Y channels of the network. This last step is done to display a random sample of every test set.

In order to compare this reproduction with the paper, the PSNR of every test set is calculated. Also, the average time it takes to produce an upsampled image by this reproduction is determined. This is computed to get an intuition about the authors claim of real-time SR of videos.

```
import random as rnd
rnd.seed(9)

# Loop over test data sets
for j in range(len(slides[1:])):

  # Load data
  test_generator = DataGenerator(slides[j+1])

  show = rnd.randint(0,len(test_generator))

  # Define lists
  outputs = []
  mse_list = []
  time_list = []

  # Main testing loop
  for i, (lr_img, bicubic, hr_img) in enumerate(test_generator):
    with torch.no_grad():

      # Only test on the Y (intensity channel)
      to_network = (lr_img[:,0,:,:]).unsqueeze(0)

      # Start timer
      start_time = time.time()

      # Run the forward pass
      outputs = ConvNet.cpu()(to_network)
      img_pred = outputs[0]

      # Substitute the Y channel from bicubic with the one outputted by the model
      bicubic[:,0,:,:] = img_pred

      #Stop timer
      elapsed_time = time.time() - start_time
      time_list.append(elapsed_time)

      # Calculate MSE
      mse = calc_mse(img_pred.numpy(), hr_img.numpy()[:,0,:,:])
      mse_list.append(mse.item())

      #Show one random image from every dataset
      if i == show:
        generate_figure(lr_img, bicubic, hr_img)

  # Printing the results
  print(slide_subfolders[j+1], ': PSNR', psnr_from_mselist(mse_list), 'Average time'
,np.mean(time_list))
```

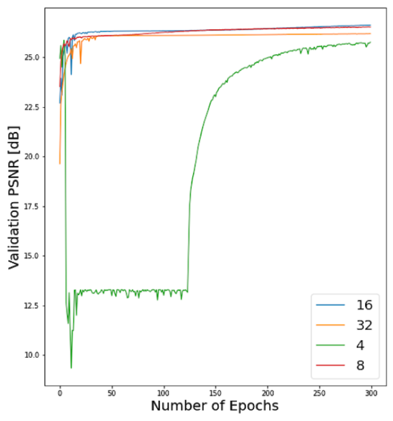
Hyperparameter tuning

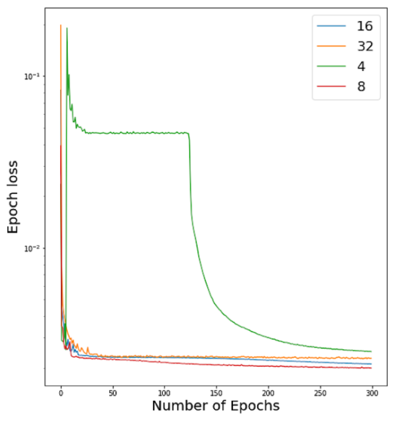
As described above there are multiple uncertainties regarding the reproducibility of the paper. Therefore, hyperparameter tuning is conducted on multiple hyperparameters in order to match the results of the paper. The hyperparameters which will have been tuned are the batch size, the learning rate threshold μ, the learning rate decay factor, the learning rate patience and the ratio between the training and the validation set. All hyperparameters are tuned independently while keeping the other hyperparameters fixed during training sessions of 300 epochs. Results of the hyperparameter tuning can be seen in the separate notebook which can be found [here]. The default values of the hyperparameters have been set to:

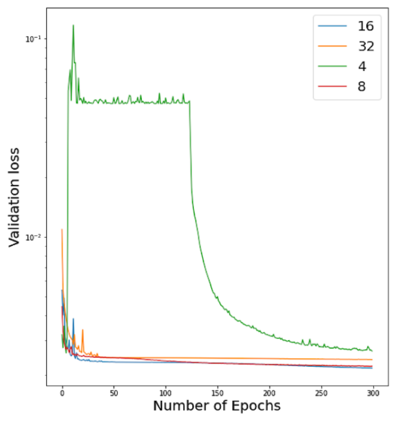
- Batch size: 16
- Learning rate threshold μ: 1e-4
- Learning rate decay factor: 0.5
- Learning rate patience: 2
- Training validation set ratio: 80/20

Batch size

Batch size is one of the hyperparameters not specified in the paper which can have an impact on the final results. Having a small batch size will increase the training speed and needs less memory but it will reduce the accuracy of gradient estimation. For optimization a batch size of 4, 8 ,16 and 32 is investigated. As it can be seen below a batch size of 16 will result in the highest performance during training. Furthermore, a strange behaviour is observed for a batch size of 4 (green line). At first, the loss decays normally, after which it suddenly explodes and decreases again. There needs to be done more research into why this is happening. The defined learning rate decay and/or Adam optimizer might be the cause of this behaviour.

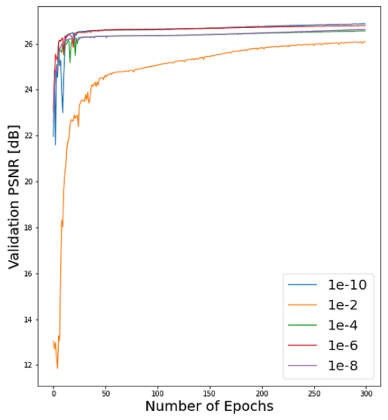


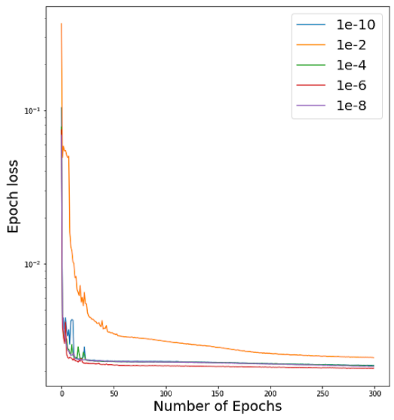


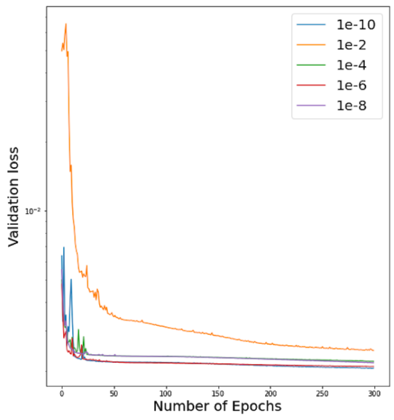


Learning rate threshold μ

The learning rate threshold μ is described as a threshold on the improvement of the cost function. If the improvement of the cost function is smaller than the learning threshold then the learning rate will be decreased. With a higher μ the learning rate will reach the final learning rate of 0.0001 earlier during training. In order to establish the μ which gives the best results a threshold of 1e-2, 1e-4, 1e-6, 1e-8 and 1e-10 is investigated. As can be seen, every threshold results in almost the same performance, with the exception of the largest threshold: 1e-2. The values 1e-6 and 1e-10 appear to give the best results, however the margin with the other values is small. Therefore, this slightly better result can be due to the randomness in the learning process (initialization of model, splitting training dataset into training and validation patches, batch generation). The experiment needs to be repeated to give a standard deviation to the curves, and thus a better answer to the question which learning rate needs to be used.

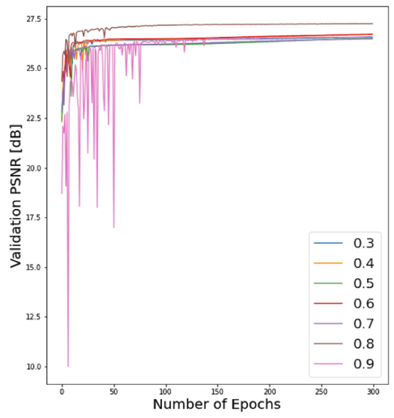


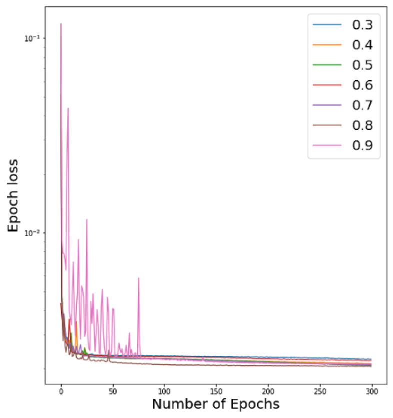


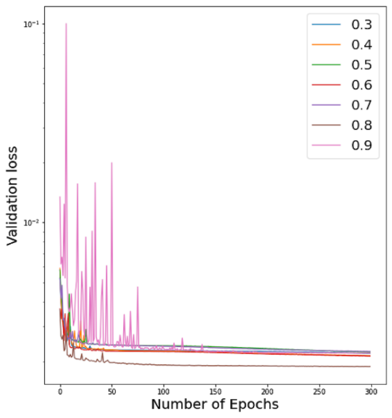


Learning rate decay factor

Whenever the learning rate threshold μ is reached, the learning rate will decrease with a certain factor. This factor is described by the learning rate decay factor and is also not specified in the paper. In contrast to the learning rate threshold μ, a higher learning rate decay factor will result in the final learning rate being reached later during training. To find the learning rate decay which results in the highest performance the model is trained on a learning rate decay factor of 0.5, 0.6, 0.7, 0.8 and 0.9. As it can be seen below a learning rate decay factor of 0.8 will result in the highest performance during training.

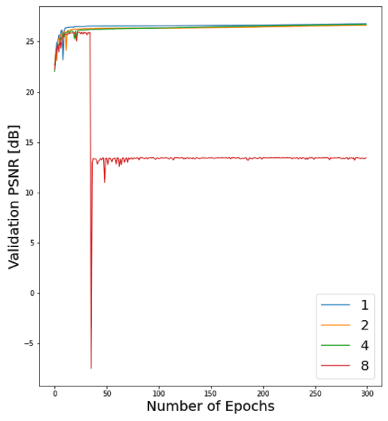


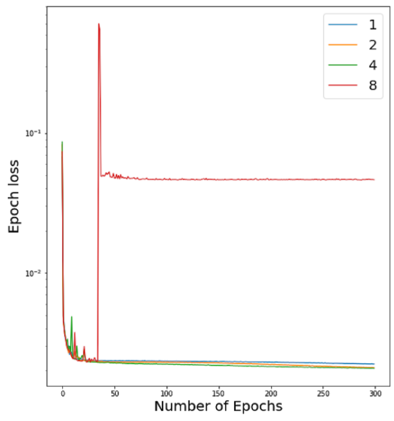


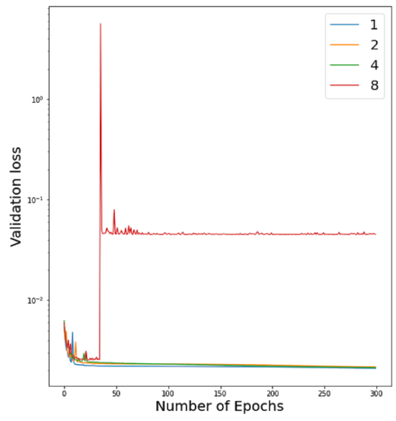


Learning rate patience

Another hyperparameter which is tunable is the learning rate patience. Whenever the learning rate threshold μ is exceeded, the model will delay the decreasing of the learning rate with a number of epochs. This delay is defined by the learning rate patience. For optimization a learning rate patience of 1, 2, 4, and 8 are investigated. As it can be seen below, a learning rate patience of 8 gives significantly lower performance compared to the others which are quite similar.

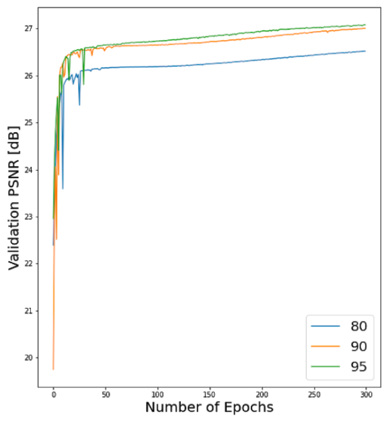


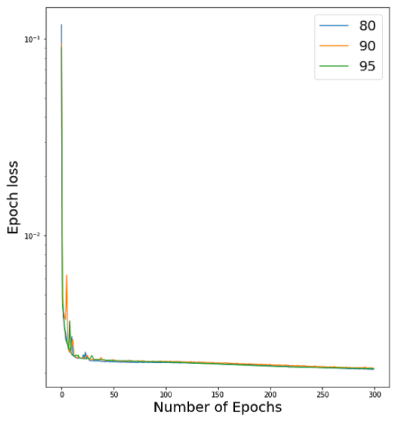


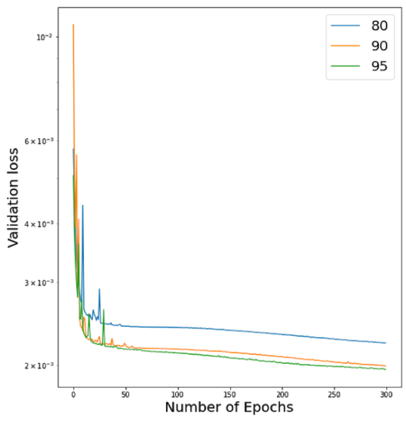


Ratio between training and validation set

The final hyperparameter which is tuned is the ratio between the size of the training set and the size of the validation set. From the yang91 training set a small subset is subsampled to validate the performance during training. For optimization usage of 80%, 90% and 95% of the training set are investigated. As it can be seen below a training validation ratio of 95/5 will result in the best performances. This is as expected, as the model has now learned on more patches per epoch and will thus be able to perform better on patches of the same types of images.







Results

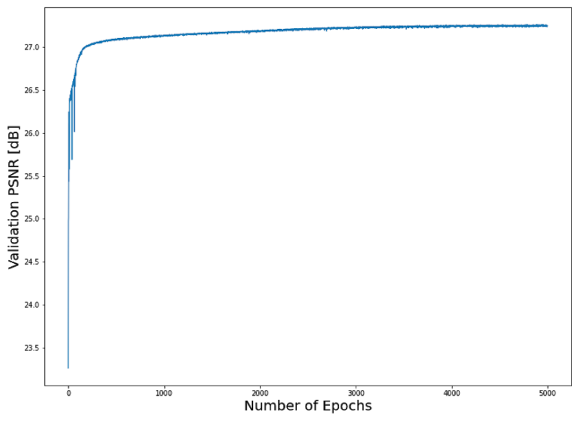
Chosen hyperparameters

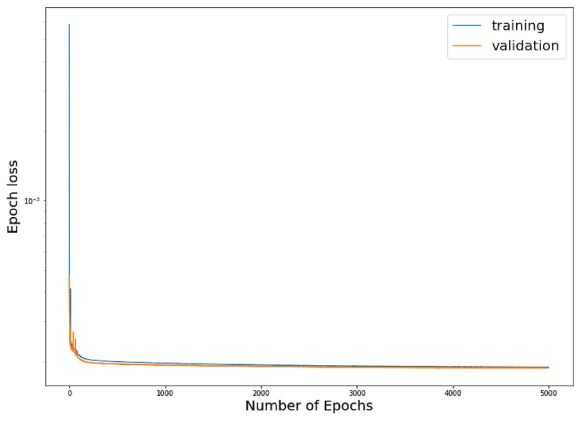
After the hyperparameter tuning, there is chosen to set the hyperparameters to the following values for the final training run:

- Batch size: 16
- Learning rate threshold μ: 1e-6
- Learning rate decay factor: 0.8
- Learning rate patience: 1
- Training validation set ratio: 95/5
- Number of epochs: 5000

Training results

In the figures below, one can see the PSNR of the validation data vs the number of epochs and the training and validation loss vs number of epochs. It can be seen that the model is learning, also for a large number of epochs. However, the amount it learns is decreasing for an increase of epochs.



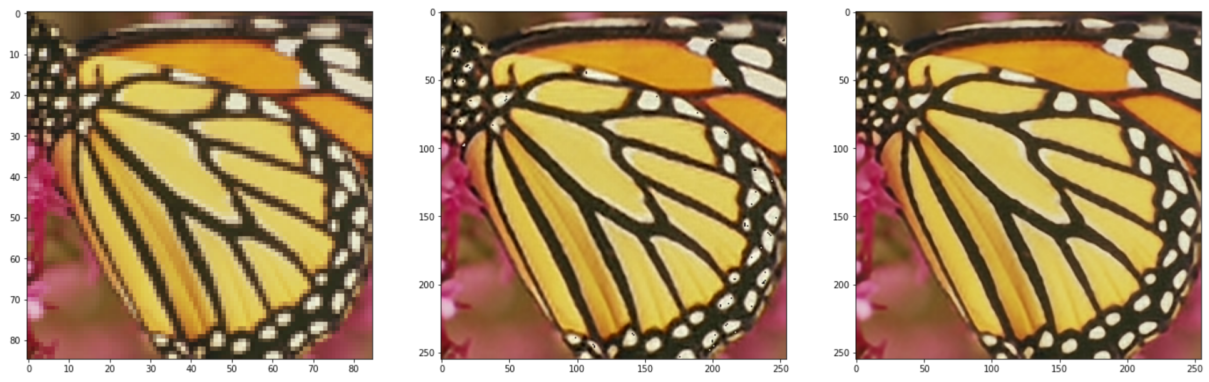


Test set results

The reproduced model has been evaluated on the 5 test datasets as described earlier. The results are shown in the table below. Values are in dB and represent the mean PSNR of that dataset.

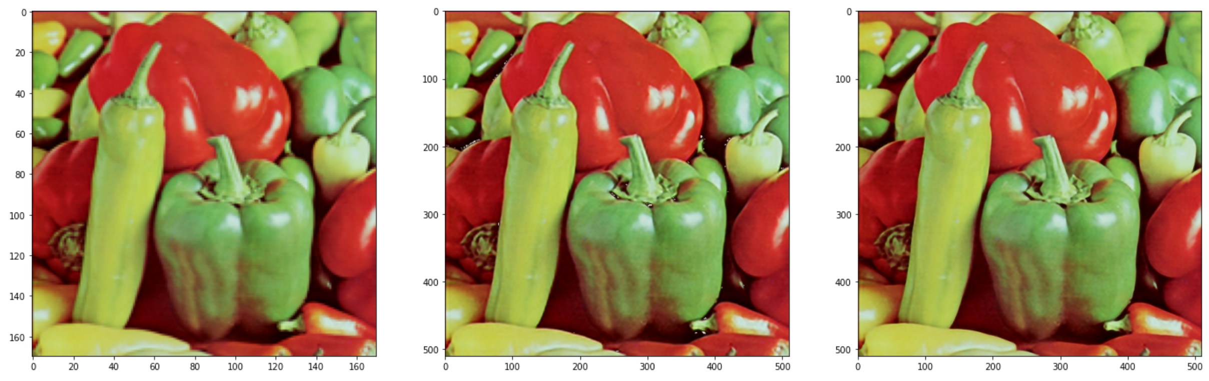
Dataset:	**Set5**	**Set14**	**BSD300**	**BSD500**	**SuperTexture136**
Results by [1], using tanh	32.55	29.08	28.26	28.34	26.42
Reproduced Results	24.15	24.10	25.29	25.16	24.40

Below you can see three images of a butterfly. The left image is the low resolution image, the middle image is the image produced by the model and the right the high resolution image.



As can be clearly seen, the model is in general capable of upsampling the low resolution image to a high resolution image. However, when looking closely at some white spots on the butterfly's wing, one can see black spots. Here, the model fails locally to produce a good image. These black spots will be discussed in the next section.

Again you can see three images below, this time from peppers (left: low resolution, middle: predicted high resolution, right: high resolution).



Again, the prediction looks to be very similar to the high resolution image. However, there appear some white spots in black areas (right bottom of image). These white spots will also be discussed in the next section.

In order to question the claim of the writers of the papers that real-time SR of 1080p videos is possible with their model, we calculated the average time it takes to produce an image by our reproduction. The obtained values in seconds are shown in the table below.

Dataset:	**Set5**	**Set14**	**BSD300**	**BSD500**	**SuperTexture136**
Time/image	0.0158	0.0303	0.0199	0.0200	0.0114

A standard video has a frame rate of 27 FPS. [5] This means that the model needs to produce an image within (1/27) 0.0370s. As can be seen in the table, this is indeed the case. However, these images are not 1080p, but more close to 180p. However, we think that the table is an indication that the model should be able to do real-time SR of 1080p videos, as the above results are obtained on a CPU of google colab. Running the model on a GPU on a good pc nowadays will most likely produce 1080p images even faster.

Discussion

Here, the reproducibility of the paper will be discussed. First, let's discuss a couple of things that were unclear about the implementation:

Unclarities

The stopping criterion
In the paper, the authors define the stopping criterion as follows: *The training stops after no improvement of the cost function is observed after 100 epochs.* However, only this sentence can already be interpreted in two ways: Train for a minimum of 100 epochs and then stop when no improvement can be observed, or, train for a number of epochs, so that a window of 100 epochs falls within some threshold value. Moreover, the value of this threshold is not mentioned in the paper. As this would require another hyperparameter to tune, it has been chosen to train the network for a fixed number of epochs, so that the network trains within some time limit. No indication on the total number of epochs has been given in the paper, so it has been set to a value of 300 during most of the hyperparameter tuning. Within 300 epochs, the loss converges enough to see if a setting works well or not. For the final model, the amount of epochs was based on available time and has been chosen to be 5000 epochs

The learning rate
The paper mentions the following about the value of the learning rate: *Initial learning rate is set to 0.01 and final learning rate is set to 0.0001 and updated gradually when the improvement of the cost function is smaller than a threshold μ.* This has been implemented with the PyTorch method: ReduceLRonPlateau, which reduces the learning rate in a similar way as described in the paper. However, the value of μ has not been mentioned in the paper. During hyperparameter tuning, several values have been tried, leading to the conclusion that only large values of μ were not working very well.

The luminance channel
The paper states the following sentence: "*For our final models, we use 50,000 randomly selected images from ImageNet [6] for the training. Following previous works, we only consider the luminance channel in YCbCr colour space in this section because humans are more sensitive to luminance changes" [7].* This sentence has been interpreted as that the model has been trained and evaluated on the luminance (Y) channel of the input images. This conclusion has been drawn by also looking at the model architecture where it seems that only one channel has been used as the input for the model. As the model also outputs the luminance channel, the choice has been made to also evaluate the model on the PSNR of two luminance images.

Discussion of our implementation:

The unclarities about the paper are not the only thing contributing to the fact that our results are different from the original paper. During the visualization of the test phase, it was noted that some of the output images contained white spots in dark areas and dark spots in light areas. We believe this is a great contributor to our apparently low PSNR. We have been trying to understand what causes this, but at the time of writing, have not succeeded in finding the cause.

Furthermore, we found some inconsistencies in the used test data. The high resolution images of one folder appear to be slightly different than the high resolution images of another folder, while this should in fact be exactly the same images. Therefore, it can be the case that we have used slightly altered test images in comparison with the paper’s test images, resulting in lower PSNR values.

To reproduce or not to reproduce?

The aim of this blogpost is to reproduce the results found in the “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network”. In order to do so, a pytorch implementation is written from scratch using only information from the paper itself. The paper presents a method for image super resolution which is fast enough so that it can be applied on real-time video material. This is done with a CNN architecture, consisting of three convolutional layers, where the feature maps are extracted on the luminance channel in low resolution space. After this, the resolution is increased only at the very end by an upsampling layer.

At the beginning of the project the paper was easy to read for us being no experts in the field of deep learning. Diving into the process of reproducing the paper we discovered that some hyperparameters for training were not specified in the paper. Also, we encountered some other uncertainties related to the stopping criterion and the luminance channel. Apart from the appearance of black and white spots we believe the visual results of our reproduction attempt are considerably accurate.

Although the performance of our reproduction is not as high as stated in the paper we believe that it is reproducible. As this was our first project in the field of deep learning we had no experience in finding the correct hyperparameters for training. For someone with a bit more experience we think the results of the paper should be easily reproducible.

References

[1] W. Shi *et al.*, “Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, doi: 10.1109/cvpr.2016.207.

[2] Y. Chen and T. Pock, “Trainable Nonlinear Reaction Diffusion: A Flexible Framework for Fast and Effective Image Restoration,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1256–1272, Jun. 2017.

[3] C. Dong, C. C. Loy, K. He, and X. Tang, “Image Super-Resolution Using Deep Convolutional Networks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2. pp. 295–307, 2016, doi: 10.1109/tpami.2015.2439281.

[4] C. Osendorfer, H. Soyer, and P. van der Smagt, “Image Super-Resolution with Fast Approximate Convolutional Sparse Coding,” *Neural Information Processing*. pp. 250–257, 2014, doi: 10.1007/978-3-319-12643-2_31.

[5] R. Taylor, “A Beginners Guide to Frame Rates : Aframe.” [Online]. Available: https://aframe.com/blog/2013/07/a-beginners-guide-to-frame-rates/. [Accessed: 20-Apr-2020].

[6] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3. pp. 211–252, 2015, doi: 10.1007/s11263-015-0816-y.

[7] S. Schulter, C. Leistner, and H. Bischof, “Fast and accurate image upscaling with super-resolution forests,” *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, doi: 10.1109/cvpr.2015.7299003.