Image Processing - 67829

Exercise 5: Neural Networks for Image Restoration

Due date: 06/02/2020

1 Overview

This exercise deals with neural networks and their application to image restoration. In this exercise you will develop a general workflow for training networks to restore corrupted images, and then apply this workflow on two different tasks: (i) image denoising, and (ii) image deblurring.

2 Background

Before you start working on the exercise it is recommended that you review the lecture slides covering neural networks, and how to implement them using the Keras framework. To recap the relevant part of the lecture: there are many possible ways to use neural networks for image restoration. The method you will implement consists of the following three steps:

1. Collect "clean" images, apply simulated random corruptions, and extract small patches.
2. Train a neural network to map from corrupted patches to clean patches.
3. Given a corrupted image, use the trained network to restore the complete image by restoring each patch separately, by applying the "ConvNet Trick" for approximating this process as learned in class.

Note: Neural networks are typically trained on GPUs, which are not available on the PCs accessible by BSc students. Without GPUs training takes much longer. The networks you will train will therefore have reduced capabilities, both in terms of quality and in terms of versatility. Nevertheless, the same code you are about to implement (up to minor adjustments) could bring you very close to the state-of-the-art

on both tasks if you use a more powerful setup. See the tips section for a workaround to help you debug your code faster, though it is purely optional.

3 Dataset handling

Training requires first building a dataset of pairs of patches comprising (i) an original, clean and sharp, image patch with (ii) a corrupted version of same patch. Given a set of images, instead of extracting all possible image patches and applying a finite set of corruptions, we will generate pairs of image patches on the fly, each time picking a random image, applying a random corruption[1], and extracting a random patch[2]. For this task you will implement the following function:

```
data_generator = load_dataset(filenames, batch_size, corruption_func, crop_size)
```

which outputs `data_generator`, a Python's generator object which outputs random tuples of the form `(source_batch, target_batch)`, where each output variable is an array of shape `(batch_size, height, width, 1)`[3], `target_batch` is made of clean images, and `source_batch` is their respective randomly corrupted version according to `corruption_func(im)`. Each image in the batch should be picked at random from the given list of filenames, loaded as a grayscale image in the [0,1] range (you should use `read_image` from previous assignments), followed by corrupting the entire image with `corruption_func(im)`, and finally randomly choosing the location of a patch the size of `crop_size`, and subtract the value 0.5 from each pixel in both the source and target image patches.

The above function has the following input arguments:

`filenames` – A list of filenames of clean images.

`batch_size` – The size of the batch of images for each iteration of Stochastic Gradient Descent.

`corruption_func` – A function receiving a numpy's array representation of an image as a single argument, and returns a randomly corrupted version of the input image.

`crop_size` – A tuple `(height, width)` specifying the crop size of the patches to extract.

Reminder: A python generator can be constructed using the following template:

```
def some_func(...):
```

[1]For the context of implementing `load_dataset` it is not critical how `corruption_func` is implemented, however, it is important to understand that using a deterministic corruption function is not useful, because it assumes we know exactly what happened to the image, and thus there is no point in learning an inverse operation.

[2]It is important that each time you will randomly select an image filename, load it, and then randomly select the location of the crop.

[3]The number 1 at the 4th axis is because we are treating the images as grayscale, and thus they have a single channel. Had we used RGB representation, then that number would have been 3.

```
while True:
    # Code for randomly creating x
    yield x
```

Important Note: since disk access could be a significant bottleneck during the training process, it is recommended to cache the images before they are cropped and corrupted. You can use a simple dictionary for this process, mapping filenames to numpy arrays of images. Before reading an image, first check if it is already in the cache and if so return it without reading it again from disk, and otherwise use the regular `read_image()` function and add the new image to the cache. While this strategy does not scale to a large number of images, it works well up to 10k-100k images (depending on image size). Additionally, it is expensive to apply a corruption function on the entire image and then only take a small crop from it. On the other hand, some corruption operations have non-local dependencies, e.g. motion blur, and so we **cannot** apply the corruption just on the random crop. As a tradeoff between the two extremes, we should instead first sample a larger random crop, of size $3 \times$ `crop_size`, apply the corruption function on it, and then take a random crop of the requested size from both original larger crop and the corrupted copy.

4 Neural Network Model

Before we can train a neural network, we first need to decide on its structure (number and type of layers). Fitting a good structure for a given task is often a very time consuming process, which requires a lot of trial and error. To save you time, we are giving you a specific network for you to use, however, after you have completed this assignment, we strongly encourage you to try and experiment yourself with different structures (adding more layers, changing the number of channels, etc.). The network you will implement is based on the ResNet architecture [1], which builds on the principle of "skip connections" – combining the outputs of nonconsecutive layers. The basic building block of ResNet is the *residual block*, which for the input X it is defined as follows: X is the input to a 3×3 convolution, followed by ReLU activation, and then another 3×3 convolution (this time no ReLU), and we denote the output of the last convolution with O. The final output of the residual block is $\text{ReLU}(O + X)$, connecting the input to the last output, skipping over the middle layers. The complete network you will implement is illustrated in the following figure:

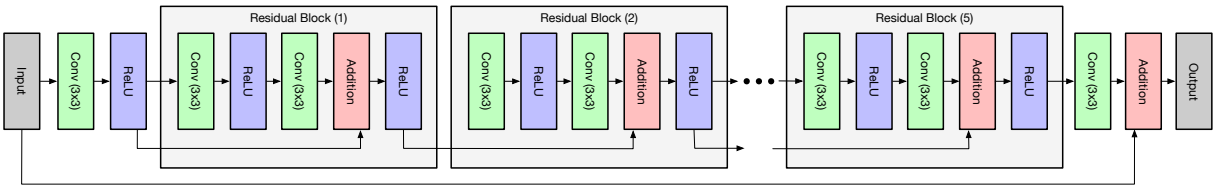


Figure 1: ResNet inspired Neural Network model for image restoration.

For implementing the above model you will use the Keras framework (with Tensorflow backend), starting with the following function for creating a residual block as described above:

```
output_tensor = resblock(input_tensor, num_channels)
```

The above function takes as input a symbolic input tensor and the number of channels for each of its convolutional layers, and returns the symbolic output tensor of the layer configuration described above. The convolutional layers should use "same" border mode, so as to not decrease the spatial dimension of the output tensor.

Next, you will implement the following function, which returns the complete neural network model:

```
model = build_nn_model(height, width, num_channels, num_res_blocks)
```

The above function should return an untrained Keras model (**do not call the compile() function!**), with input dimension the shape of `(height, width, 1)`[4], and all convolutional layers (including residual blocks) with number of output channels equal to `num_channels`, except the very last convolutional layer which should have a single output channel. The number of residual blocks should be equal to `num_res_blocks`.

Allowable classes/methods: you can only use the the following Keras' classes/methods to construct the model: `Model, Conv2D, Activation, Input, Add`. You are not allowed to use the Sequential API!

Note: you are allowed to loop over the number of residual blocks when creating the model.

5 Training Networks for Image Restoration

Now that we have a dataset and neural network, it is time to put everything together and train it.

```
train_model(model, images, corruption_func, batch_size,
            steps_per_epoch, num_epochs, num_valid_samples)
```

The above function should divide the images into a training set and validation set, using an 80-20 split, and generate from each set a dataset with the given batch size and corruption function (using the function

[4]The "1" at the end is because the network expects grayscale images. If it were RGB then it would have been "3".

from section 3). Then, you should call to the `compile()` method of the model using the "mean squared error" loss and ADAM optimizer. Instead of the default values for ADAM, import the class `Adam` from `keras.optimizers`, and use `Adam(beta_2=0.9)`. Finally, you should call `fit_generator` to actually train the model.

The input arguments of the above function are:

`model` – a general neural network model for image restoration.

`images` – a list of file paths pointing to image files. You should assume these paths are complete, and should append anything to them.

`corruption_func` – same as described in section 3.

`batch_size` – the size of the batch of examples for each iteration of SGD.

`steps_per_epoch` – The number of update steps in each epoch.

`num_epochs` – The number of epochs for which the optimization will run.

`num_valid_samples` – The number of samples in the validation set to test on after every epoch.

Up to this point we have trained a neural network model on a given training set. Next, we move to implementing the prediction step for restoring images.

6 Image Restoration of Complete Images

The network we learn with `train_model` can only be used to restore small patches. Now we will use this base model to restore full images of any size. As explained in class, we will use a property of convolutional networks to "stretch" our trained model to any given image. We do so by creating a new network for each given image (see note at the end of section). We then simply use the new model to restore the given full image.

You should implement the following method according to the above discussion:

```
restored_image = restore_image(corrupted_image, base_model)
```

You will need to create a new model that fits the size of the input image and has the same weights as the given base model, before using the `predict()` method to restore the image. Finally, you will need to clip the results to the $[0, 1]$ range. The above function has the following input arguments:

`corrupted_image` – a grayscale image of shape `(height, width)` and with values in the $[0, 1]$ range of type float64 (as returned by calling to `read_image` from ex1, that is affected by a corruption generated from the same corruption function encountered during training (the image is not necessarily from the training set though). You can assume the size of the image is at least as large as the size of the image patches during training.

`base_model` – a neural network trained to restore small patches (the model described in section 4, after being trained in section 5). The input and output of the network are images with values in the $[-0.5, 0.5]$ range (remember we subtracted 0.5 in the dataset generation). You should take this into account when preprocessing the images.

Note: to adjust the model to the new size, you can simply create an input tensor the size of the input image, and then apply the base model to this input tensor, and then create a new adjusted model, as follows:

```
a = Input(...) # with width and height of the image to be restored.
b = base_model(a)
new_model = Model(inputs=a, outputs=b)
```

7 Application to Image Denoising and Deblurring

Finally, you will utilize the above workflow to solve two separate tasks: (i) Image Denoising, and (ii) Image Deblurring. In practice, you could have solved both these tasks at once, however, due to the computational limitations we have, we focus instead on solving each task separately.

7.1 Image Denoising

In this section you will learn a denoising algorithm for removing i.i.d. additive gaussian noise of unknown standard deviation made for natural images.

7.1.1 Gaussian Noise

For the task of image denoising, you will first need to implement the following random noise function for training and then testing your model:

```
corrupted = add_gaussian_noise(image, min_sigma, max_sigma)
```

Which should randomly sample a value of `sigma`, uniformly distributed between `min_sigma` and `max_sigma`, followed by adding to every pixel of the input image a zero-mean gaussian random variable with standard deviation equal to `sigma`. Before returning the results, the values should be rounded to the nearest fraction $\frac{i}{255}$ and clipped to $[0, 1]$. You can use `numpy.random.normal` to general the required sample in the shape of the image. The input arguments to the function are:

`image` – a grayscale image with values in the $[0, 1]$ range of type float64.

`min_sigma` – a non-negative scalar value representing the minimal variance of the gaussian distribution.

`max_sigma` – a non-negative scalar value larger than or equal to `min_sigma`, representing the maximal variance of the gaussian distribution.

7.1.2 Training a Denoising Model

Given the function from the previous section, you should implement the following function which will return a trained denoising model:

```
model = learn_denoising_model(num_res_blocks=5, quick_mode=False)
```

Use the function `sol5_utils.images_for_denoising()` we supplied for you to get the image paths list for training on this task. You can download the actual images from Moodle, and place them in `image_dataset` directory. The above method should train a network which expect patches of size 24×24, using 48 channels for all but the last layer. The corruption you will use is a gaussian noise with sigma in the range $[0, 0.2]$ – you can wrap the function `add_gaussian_noise` in a lambda expression to fit the protocol of `corruption_func`, or alternatively use a local function. Finally, use 100 images in a batch, 100 steps per epoch, 5 epochs overall and 1000 samples for testing on the validation set. For debugging purposes, you can use fewer epochs to make sure you model partially works, before taking the time to train the full model.

The above function has a single argument used solely for the presubmission phase. Because it's not possible to fully train your model in the presubmission script we use, we require you to add a special quick mode for the above function for testing purposes. If `quick_mode` equals `True`, instead of the above arguments, use only 10 images in a batch, 3 steps per epoch, just 2 epochs and only 30 samples for the validation set. Under this mode your function should finish in a few seconds at most.

Important: Do not include in your submission the images in the dataset – we will place them in the correct directory for your code to run (this is why we provided you with the helper function).

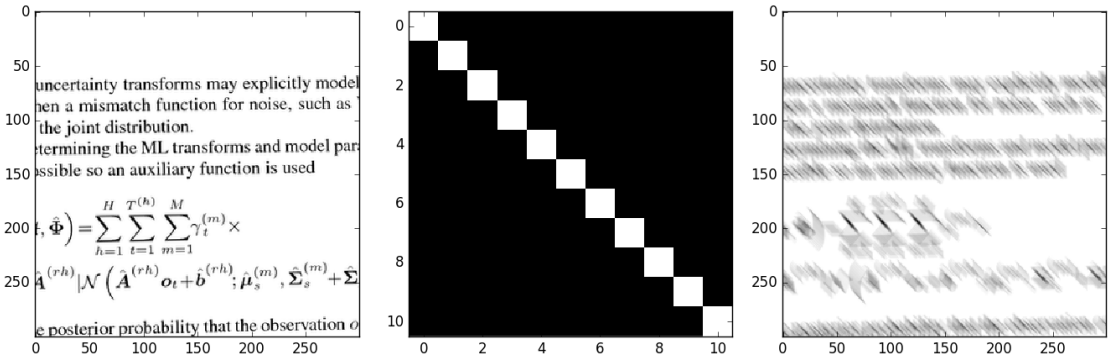
7.2 Image Deblurring

In this section you will learn a deblurring algorithm for correcting motion blurred images of text, e.g. the case where you wish to use your mobile phone to take a picture of a document while your hand is slightly shaking.

7.2.1 Motion Blur

For the task of image deblurring, we will focus on motion blur applied to images of text. Motion blur can be simulated by convolving the image with a kernel made of a single line crossing its center, where

the direction of the motion blur is given by the angle of the line. See the following example of an image of text, motion blur kernel, and its result:



For the assignment you are required to implement the motion blur function described above, and its random variant:

```
corrupted = add_motion_blur(image, kernel_size, angle)
corrupted = random_motion_blur(image, list_of_kernel_sizes)
```

The function `add_motion_blur` should simulate motion blur on the given `image` using a square kernel of size `kernel_size` where the line (as described above) has the given `angle` in radians, measured relative to the positive horizontal axis, e.g. a horizontal line would have a zero angle, and for the figure above the angle would be $3\pi/4$ (or $135°$ in degrees). To simplify this assignment, we provide you with the function `sol5_utils.motion_blur_kernel(kernel_size, angle)` for generating this kernel, which you can then apply to the image with `scipy.ndimage.filters.convolve`. The function `random_motion_blur(image, list_of_kernel_sizes)` samples an angle at uniform from the range $[0, \pi)$, and choses a kernel size at uniform from the list `list_of_kernel_sizes`, followed by applying the previous function with the given image and the randomly sampled parameters. Before returning the image it should be rounded to the nearest fraction $\frac{i}{255}$ and clipped to $[0, 1]$. The input arguments to the above two functions are:

`image` – a grayscale image with values in the $[0, 1]$ range of type float64.

`kernel_size` – an odd integer specifying the size of the kernel (even integers are ill-defined).

`list_of_kernel_sizes` – a list of odd integers.

`angle` – an angle in radians in the range $[0, \pi)$.

7.2.2 Training a Deblurring Model

Given the function from the previous section, you should implement the following function which will return a trained deblurring model:

```
model = learn_deblurring_model(num_res_blocks=5, quick_mode=False)
```

Use the function `sol5_utils.images_for_deblurring()` we supplied for you to get the image paths list for training on this task. You can download the actual images from Moodle, and place them in `text_dataset` directory. The above method should train a network which expect patches of size 16×16, and have 32 channels in all layers except the last. The dataset should use a random motion blur of kernel size equal to 7 – you can wrap the function `random_motion_blur` in a lambda expression to fit the protocol of `corruption_func`, or alternatively use a local function. Finally, use 100 images in a batch, 100 steps per epoch, 10 epochs overall and 1000 samples for testing on the validation set. For debugging purposes, you can use fewer epochs to make sure you model partially works, before taking the time to train the full model.

The above function has a single argument used solely for the presubmission phase. Because it's not possible to fully train your model in the presubmission script we use, we require you to add a special quick mode for the above function for testing purposes. If `quick_mode` equals `True`, instead of the above arguments, use only 10 images in a batch, 3 steps per epoch, just 2 epochs and only 30 samples for the validation set. Under this mode your function should finish in a few seconds at most.

Important 1: Do not include in your submission the images in the dataset – we will place them in the correct directory for your code to run (this is why we provided you with the helper function).

Important 2: Notice we use slightly different configurations from the training of the denoising model.

7.3 Effect of depth

Use the `num_res_blocks` parameter to test models of different depths for both of the above tasks. Produce a plot of the mean square error on the validation set with respect to the number of residual blocks (from 1 to 5), and add it to your submission as `depth_plot_denoise.png` and `depth_plot_deblur.png`. Finally, answer the following question:

Q1: Describe the effect of increasing the residual blocks on its performance for each task, both quantitatively in terms of the plot you got and qualitatively in terms of the differences in the image outputs of each model.

7.4 Image Super-resolution

The task of super-resolution is to take a low-res image and estimate its high-res counterpart. The workflow you have implemented in this assignment can be used to solve this task as well. While you are not required to implement this use-case your self, you will need to answer the following question:

Q2: suggest how to use the workflow you have implemented in this assignment to train a network that could be used for super-resolution. A full answer consists of a description of the corruption function, and how to properly use the `restore_image()` function for this task. For this context only, you could optionally add additional operations before calling to `restore_image()`.

Note: you are encouraged to also implement this method for your self, however, from our testing it takes at least a couple hours of training to get to good results, which is why you do not need to add it to your submission.

8 Bonus (+10): Deep Image Prior

In the previous sections we have used the machine learning framework to learn models that can solve image restoration tasks by directly modeling the corruption process. In the recently published article "Deep Image Prior" by Ulyanov et al., the authors have demonstrated that the very structure of deep convolutional networks serves as a good prior for what natural images look like, and that we can employ without any learning to restore images without any knowledge on the corruption process.

For this section, you should read the article (you can download it from here: `https://arxiv.org/abs/1711.10925`), and implement the alternative method they propose for image restoration. Specifically, implement the method described in the sub section titled "Denoising and generic reconstruction" with the following signature:

```
restored_image = deep_prior_restore_image(corrupted_image)
```

In your function you are free to use either the models used in the rest of this assignment or experiment with your own models (or ones suggested in the article). Due to computational constraints, you can assume that we will test this function with images which are exactly 64-by-64 pixels. Naturally, due to runtime constraints you might not be able to get to results as good as the ones presented by the authors, however, you should be able to reproduce the underlying effect.

9 Tips & Guidelines

- Do not forget to answer **Q1** from section 7.3, and to produce the required plots. Write your answer in `answer_q1.txt`.
- Do not forget to answer **Q2** from section 7.4. Write your answer in a text file named `answer_q2.txt`.
- You are only allowed to use the Tensorflow. Keras framework for implementing neural networks. Do not use "vanila" Tensorflow directly. It is okay to use additional functions when implementing the bonus.
- Do not attach the image datasets we provide to your submissions. We will make sure the datasets are in place when running your code.
- Be sure to implement quick mode in such a way that testing under this configuration use the exact same methods as the regular mode with the exception of different parameters. We suggest you define the parameters using local variables conditionally on `quick_mode`, and then use them throughout the rest of your learning methods.
- You can assume **legal input** to all functions.
- All input images are represented by a matrix of class `np.float64`.
- Avoid unnecessary loops in your code, e.g. for pixel-wise operations. You can loop over images in a batch in the `load_dataset` function. You can loop over number of residual blocks when building the model.
- Though this assignment was designed with a CPU constraint in mind, it could still take a long while to train each model (15 min based on our measurements). To help you debug your code, and offer you a place to experiment with larger network architectures, you can (optionally) load and test your code on Google's Colab. We provide an example notebook on how to load/import python scripts directly from your Google's Drive directory, for easier testing.
  Open: `https://colab.research.google.com/drive/11Mzm9MVrmqDMS-Jg0l0mQ4DX77BKZf3s`
- Following the above point, if you want to see what this model is capable of, try using a model with 10 residual blocks, using 64x64 crops, and trained for 100 epochs (or something on that order). Such a model can produce very good results that are on par with classical human-designed methods. To go beyond that requires additional training methods that are beyond the scope of this course.

10 Submission

Submission instructions may be found in the ”Exercise Guideline‘s” document published on the course web page. Please read and follow them carefully.
Good luck and enjoy!

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Computer Vision and Pattern Recognition*, pages 770–778, 2016.