Rendering Natural Camera Bokeh Effect

Using the PyNET Convolutional Neural Network

Harsh Poonia
2nd Year UG, CSE

Report for the project under Winter in Data Science,
Analytics Club, IIT Bombay
Mentored by Vinayak Srivastava

January 2023

Rendering Natural Camera Bokeh Effect

Using the PyNET Convolutional Neiral Network

Harsh Poonia

Abstract

Bokeh is an important artistic effect used to highlight the main object of interest on the photo by blurring all out-of-focus areas. While DSLR and system camera lenses can render this effect naturally, mobile cameras are unable to produce shallow depth-of-field photos due to a very small aperture diameter of their optics. Unlike the current solutions simulating bokeh by applying Gaussian blur to image background, in this implementation of the paper conceptualised by the Andrey Ignatov and team at ETH Zurich Vision Lab, we try to learn a realistic shallow focus technique directly from the photos produced by DSLR cameras.

The architecture of the model has been taken from `github.com/aiff22/PyNET-bokeh`, but the code served only to guide us in the journey, for it was written in Tensorflow 1.x. The model backend has been written in Tensorflow 2.x and Keras, and the data input pipeline was created from scratch in tensorflow. A subset, more than one fifth of the original *Everything is Better with Bokeh!* dataset was used for training, for some hours on a cloud based NVIDIA Tesla P100 GPU.

While the model is able to reasonably predict and apply the bokeh effect, limitations in training resources did not allow the model to learn colors in the dataset, so the output spectrum has little color contrast.

The code for the repository can be found at my github below.

harpoonix

PyNET Architecture

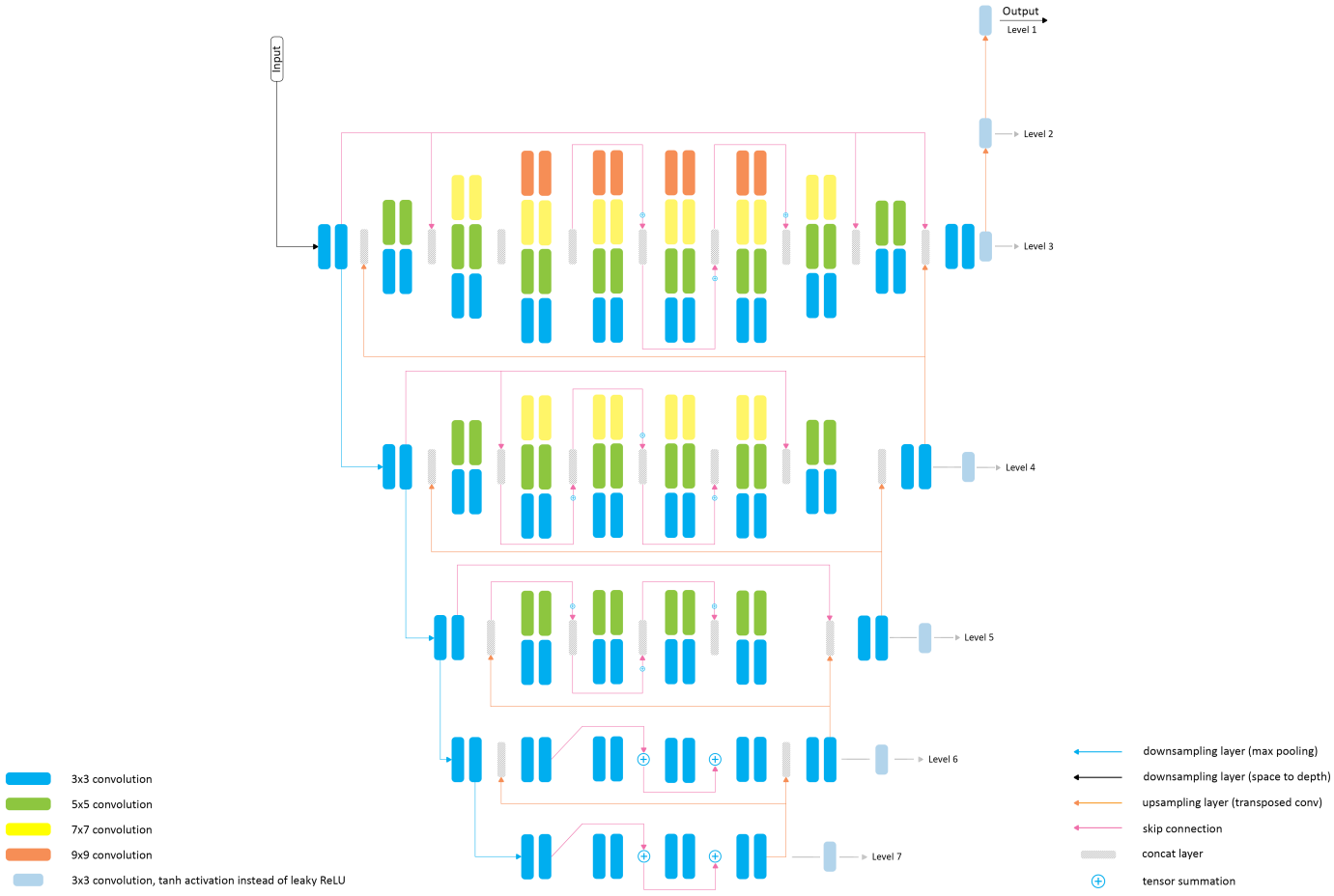

Figure 1: The PyNET CNN

The model has an inverted pyramidal shape and is processing the images at seven different scales. The model is trained from the bottom up - starting from level 7 and moving our way upwards. This allows us to obtain good recontruction results at lower layers that work with low resolution images, and each higher level gets upsclaced high quality feautres from lower levels.
At each convolutional layer, we perform instance normalisation on each of the channels to prevent training from diverging. Each color channel is normalised across its height and width while training. Skip connections and upsampling through transpose convolutional layers allow better learning of low level and high resolution features.

Data Input Pipeline

Getting the paired image dataset

We use `tf.data.Dataset` API and asscoiated methods for building the custom input pipeline for our CNN. We can create a Dataset object from both the `original` and `bokeh` folders by picking files that match a particular regex. We zip these datsets together to get a dataset made up of original and bokeh image pairs. This dataset needs to be processed to convert file names to image numpy arrays / tensors for feeding into the ConvNet. The conversion can be done through `tf.io.decode_jpeg` method.

Preprocessing

The input to the model is of dimensions $512 \times 512 \times 3$ (512 pixels height and width, 3 color channels (RGB)). The images in the dattaset are of height 1024px and width any unmber around $1500 \sim 1600$px. We first resize the width to be a multiple of 32, so we choose 1536px for uniformity. Now we need to extract random square crops of 1024×1024px for the bokeh images, and downsize the corresponding crop in the original to 512×512px to feed into the neural network. We use the `num_parallel_calls` parameter of the map function to optimise the proprocessing steps on the entire dataset. The preprocessed dataset contains (`original`, `bokeh`) pairs of images where original is 512px and bokeh is 1024 px.

Batch Size, Shuffling and Prefetching

Gradient Descent on the average of the desired nudges to the weights by each sample can be slow, so we instead divide the dataset into **mini-batches**, compute the gradient descent step on each batch and thus speed up the training. The recommended batch size for the paper is $8 \sim 50$, but due to memory limitations, the 16GB Tesla P100 could only fit a batch size of 4. This lead to some **unstable training**. The dataset can be **shuffled** in the beginning, a buffer of 1000 was maintained. **Prefetching** the next batch means loading and getting the next batch ready on the GPU while one batch is being trained. This saves time and speeds up training on GPUs.

Training the Model

Loss Function

The loss function used for the first level of output was a combination of the L_1 loss (1-norm of the difference between the output and the target bokeh image), the SSIM (Structural Similarity Index) loss and perceptual VGG-based loss function. The last part is the absolute difference between the outputs produced by a very deep pretrained VGG-19 neural network when input is the target bokeh image and the output.

$$\mathscr{L} = \mathscr{L}_1 + \mathscr{L}_{SSIM} + 0.01\mathscr{L}_{VGG}$$

Optimizers

We use the momentum-based gradient descent combined with RMSProp, aka the Adam optimiser, with a learning rate of 5×10^{-5}. The model was trained in multiple sessions, so a callback is set up that saves the best model version after every epoch so that learning can resume from where we left off.