See in the dark

Deep Learning Homework Paper

Takáts Bálint
Mészáros Gergő
Oszvald Levente

Abstract

Taking pictures in a low light environment can be challenging, for many reasons. Short exposure images tend to suffer from noise, while long exposure images are suffering from blurriness. Several techniques have been introduced to help reduce noise and blurriness, but none of them were perfect. Our team has decided to reproduce the "Learning to See in the Dark" paper from 2018, which was a big step towards improving low light photography. We chose to optimize the algorithm by using a smaller, but hopefully faster fully connected neural network, with U-net architecture, featuring batch-normalization, and in addition we created another convolutional neural network capable of predicting each images' amplification ratio, based on the image itself as an input.

Kivonat

Fényképezés gyenge fényviszonyok között több okból kifolyólag is akadályokba ütközhet. Rövidebb záridő esetén a fényképünk nagy eséllyel zajos lesz, hosszabb záridő esetén pedig elmosódás lesz megfigyelhető. Az idők során több technikai újítás is érkezett, amik ezt a problémát voltak hivatottak orvosolni, de nem nagy sikerrel. A csapatunk egy 2018-ban kijött, "Learning to See in the Dark" című értekezés - ami megjelenésekor nagy lépés volt rossz fényviszonyok közötti fényképezés javítására - átalakítását tűzte ki célul. Tervünk, az előbb említett projekt algoritmusának optimalizálása egy kisebb, de remélhetőleg gyorsabb neurális hálózattal, ami a U-net architektúra alapjaira épül. Továbbá egy másik, konvolúciós neurális hálózatot is létrehoztunk, amely képes egy rossz fényviszonyok között készült képről - mint bemenetről - megállapítani annak amplifikációs értéket(amplification ratio).

Introduction

Imaging in low light circumstances is challenging. Using a higher ISO brightens the picture, however introduces a lot of noise, and using a wider aperture is often not possible since it is hardware constrained. The team in 2018 created a freely available dataset, consisting of more than 5000 images. Since these images are raw, the size of the compressed dataset is over 75 GB. There were two different datasets, Sony and Fuji, so we have chosen to go with the Sony dataset, which is more than 60 GB uncompressed. The images are not really interesting, they were taken in low light environments, with the exposure being between 1/10 and 1/30 of a second. For every low light image there is a given ground truth image(to be exact, for every 12 low light images there is a ground truth one) with an exposure between 10 and 30 sec. Meaning a 100 to 300 longer exposure(or 7 to 9 steps). The images are 4240×2832 pixels, which means passing through a whole image would be very memory intensive. Our team decided to use smaller patches and pass those through the network, leading to big improvements in time.

Related work

The idea of the project came from a paper published in 2018, called "Learning to See in the Dark", done by four computer scientists. Their paper was a big step towards improving low light photography. After a few months of their release, Google came out with Night Mode, and several manufacturers followed. The Dataset they used - as previously mentioned - combined of a Sony and a Fuji dataset. Their network used two different fully convolutional networks: the Sony dataset featuring CAN(mainly used for fast image processing), and the Fuji dataset featuring U-net. They used patches, sized 512x512, and performed random flipping for data enrichment. The paper leaves room for improvement. One of the problems of their model is that the amplification ratio must be chosen externally. The amplification ratio scales the brightness of the image to a desired value. So it would be useful to infer it from the input image.

Environment

We are using Jupyter Notebook, since a lot of visualizations are being shown through the coding phase. Due to hardware(mostly GPU and RAM) limitations, we set up a Google Cloud instance, featuring 1 * Nvidia Tesla K80 GPU, 500 GB SSD, n1-standard-2 (2 vCPUs, 7.5 GB memory), Ubuntu 18.04 LTS, and the zone being europe-west1-b. We installed CUDA, which is necessary for general computing on GPUs. Setting up the GPU environment did not go as planned, but after re-setup, GPU was available. For training we resized the available memory to 75 GB, which is the maximum memory allowed with the K80 GPU. Even with this much memory we were not able to load the whole dataset into memory so we only trained on 20% of the images at a time.

Network

Our project features two different neural networks for each task, both layers are convolutional:
First, the network for predicting the correct amplification ratio of a given raw input image. This network is a basic fully convolutional network featuring 2d convolutional, maxpooling, one flatten and finally a dense layer. The goal is to "shrink" the 3d image into one number corresponding to the required output value. The input image fed into this pipeline has to be 64x64x3 pixels. We use MSE(Mean Squared Error) to measure loss, since we would like to know the difference between the required output value and the prediction.
Second, the network trained on the raw images. This network is based on the U-net architecture. U-net is a convolutional network, consisting of a contracting path and an expansive path. The contracting path is based on 2d convolutional and maxpooling layers, meaning it works like an encoder, reducing the width, height of the input image. The expansive path is a sequence of up-convolutions and concatenation with high resolution features from the contracting path, which leads to a high-resolution output image. We tried to

somehow improve the speed of this network by adding Batch Normalization, and Dropout. About the hyperparameters, for optimizer we went with Adam, with a learning rate of 0.001

Implementation

Collecting the data, preparation

As above mentioned, our dataset consists of images taken by the group that wrote the reference paper. We have approximately 2500 images - each image with its raw and ground truth image pair. The resolutions are 4240×2832, thus the dataset is more than 60 GBs. After unzipping, the images are divided into three directories, train, valid and test, in these directories each line contains four texts, the path to the input image, the path to its ground truth image, the ISO score, and aperture. Then the raw images are loaded into the correct 2-dimensional arrays with their ground truth pair.
The two networks require different data preparation, described below.
The convolutional network predicting the correct amplification ratio uses 64x64x3 images as input images. Since the raw images have way higher resolution, we resized the input raw images to fit the network.(For this we created a script that is capable of resizing the images to way smaller resolutions, using the cv2 library) The network uses amplification ratio values as desired outputs, so after normalization we store these in a set as well. We created six arrays, an input and an output pair for train, validation, and test, which later are fed into the cnn.
Our U-net based network

Training

Training on the smaller fully convolutional network was way faster than the other network. After the preparation of the data, we used the prepared training and validation input, output to measure the strength of the network. We used size 16 for batches, and the training lasted for 100 epochs. For callback functions, we used early-stopping monitoring the validation loss, to reduce the chances of overfitting. In addition, ReduceLROnPlateau was used to help reduce learning rate, if validation loss does not improve over a given number of epochs (patience). At first, the network could not produce efficient outputs, loss and validation loss could not improve but decrease, even after 100 epochs, predictions were improper. To improve, we tried hyperparameter optimization. Training on the larger network was an immense task. We struggled with it quite a lot, because we have had no experience with this large datasets before. We only had access to a vm with 75GB worth of memory at most, so we had to make do with that. So we trained the model with only 20% of the data at a time so it could fit into memory.

Evaluation, Hyperparameter optimization

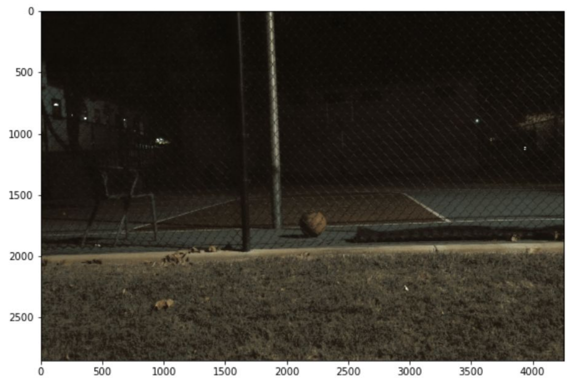
The network, trained on the 64x64x3 images - as mentioned - could not improve, so optimizing the hyperparameters were necessary. We created a script for this process; using Hyperas, a simple wrapper around hyperopt for fast prototyping around Keras models. Following the Hyperas conventions, while building the model, every hyperparam had different options to choose from, layer feature sizes, the rate of dropout, type of optimizer, activation function, the value of learning rate, etc. At first, we tried training on 25 epochs for one hyperparam set, and this method over 50 evaluations. The optimization lasted for approximately 30 minutes, so even higher number of evals is in an acceptable range. The best validation loss we achieved was 0.447, meaning with a small optimization, improvements were done.

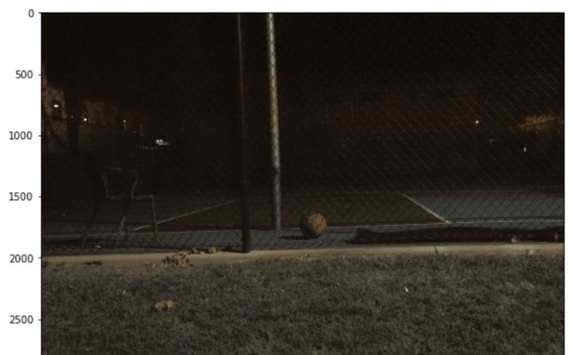
Test

To start, we tested the smaller network's efficiency. The test dataset was prepared as the train and validation set, test input resized, output rescaled. The testing was done after optimizing the hyperparameters. The results were quite close to their ground truth pair, to be even more exact, we calculated their (abs(log2(prediction) - log2(truth))) value to get a better picture; here are some examples:

Predicted ratio	True ratio	Divergence(equation above)
262.9287	300	0.1902904777798966
270.1934	300	0.1509699572833689
245.1514	250	0.0282551078528706

We suppose the small difference between the ratios is due to the resizing of the images, since a lot of information got lost in that process.

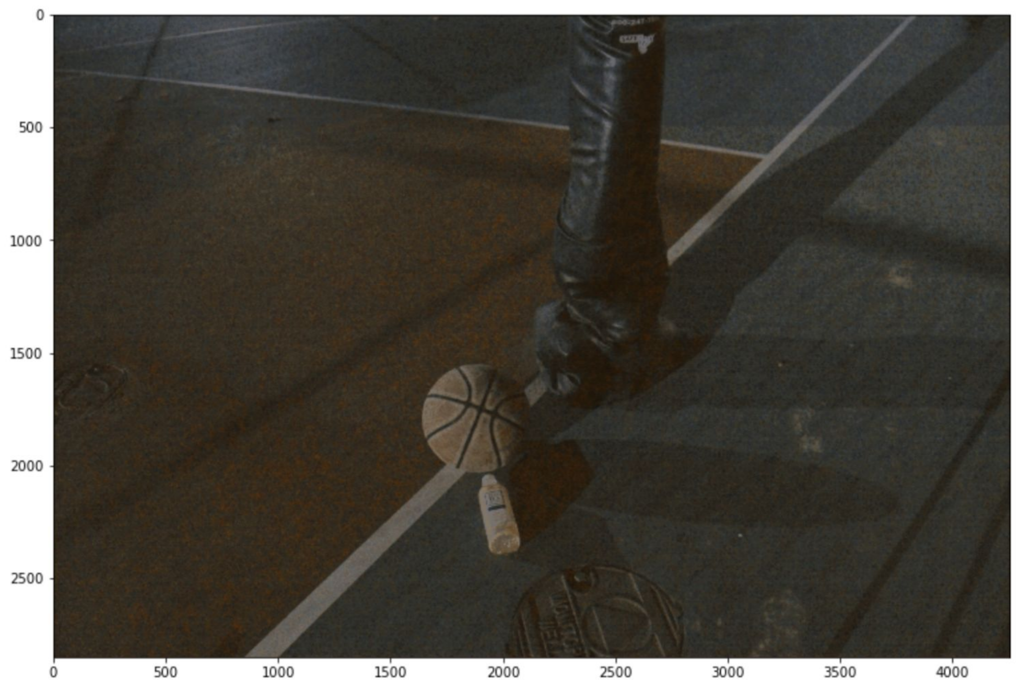




We ran the two algorithms together, and we can see the results on the left. The image on top was created using the ground truth amplification ratio. The image on the bottom was created 100% percent with learning based models.

They are clearly very similar. This is because even seemingly large differences in the amplification ratio produce the same results. In photography it is not uncommon, even among professional photographers, to miss the exposure by a stop.

During training we sometimes run the algorithm on an image from the test dataset. This image was captured after we trained the network on the whole training data for 20 epochs. 20 epoch is not sufficient to train such a large network unfortunately, however one can clearly see how much it has learned.



Testing the larger network after training was a challenge as well. Due to memory and compute limitations we were not able to run the algorithm on the full test data set, thus we selected 10 images randomly, randomly cropped them and ran the evaluation. We measured an MSE of 0.015 which corresponds to an error of 31 in the RGB image space. Meaning for example if the truth is 100 the network on average guesses between 70 and 130. This is a large error margin, clearly not enough for a commercial application. On the other hand we achieved this level of accuracy with only ~20 epochs worth of training. We are quite certain that with more resources we would be able to reduce the MSE error by tenfold, because during training we saw the MSE loss dip below 0.002. One can find more images in the jupyter notebooks.

Android App

To make the predictions more sightful, we came up with an Android Application, which is capable of - through user interaction - showing how the RAW images improve over the training. The implementation of this application was not easy, since we had to use a Pyhton SDK called Chaquopy, which lets you intermix Python and Kotlin. The main logic behind the app was to let the user choose one from three options, and then shows the ground truth image, and above the predicted image. Since Chaquopy does not feature rawpy yet, we had to transform the .ARW images to .JPEG, but that did not affect the training at all. Due to lack of time, we ran into one problem we could not get through. Loading in our pretrained model could not be done, since the path was not pointing to a file or directory. At first, we thought that this problem is mainly due to this SDK, so we tested the code in Pycharm environment, where both the loading in, and the prediction happened to be working. Finding this issue, and solving it will be a future task.

Future plans, Summary

One promising plan is to make our application capable of taking the actual image, not just showing the prediction.

References

1. https://cchen156.github.io/paper/18CVPR_SID.pdf
2. https://arxiv.org/pdf/1709.00643.pdf
3.