Deep CNN-Based Blind Image Quality Predictor in Python

Ricardo Ocampo Follow
Oct 19 · 8 min read

Introduction

In this tutorial, we will implement the *Deep CNN-Based Blind Image Quality Predictor (DIQA)* methodology proposed by Jongio Kim, Anh-Duc Nguyen, and Sanghoon Lee [1]. Also, I will go through the following TensorFlow 2.0 concepts:

- Download and prepare a dataset using a *tf.data.Dataset* builder.
- Define a TensorFlow input pipeline to pre-process the dataset records using the *tf.data* API.
- Create the CNN model using the *tf.keras* functional API.
- Define a custom training loop for the objective error map model.
- Train the objective error map and subjective score model.
- Use the trained subjective score model to make predictions.

Note: Some of the functions are implemented in utils.py as they are out of the guide's scope.

What is DIQA?

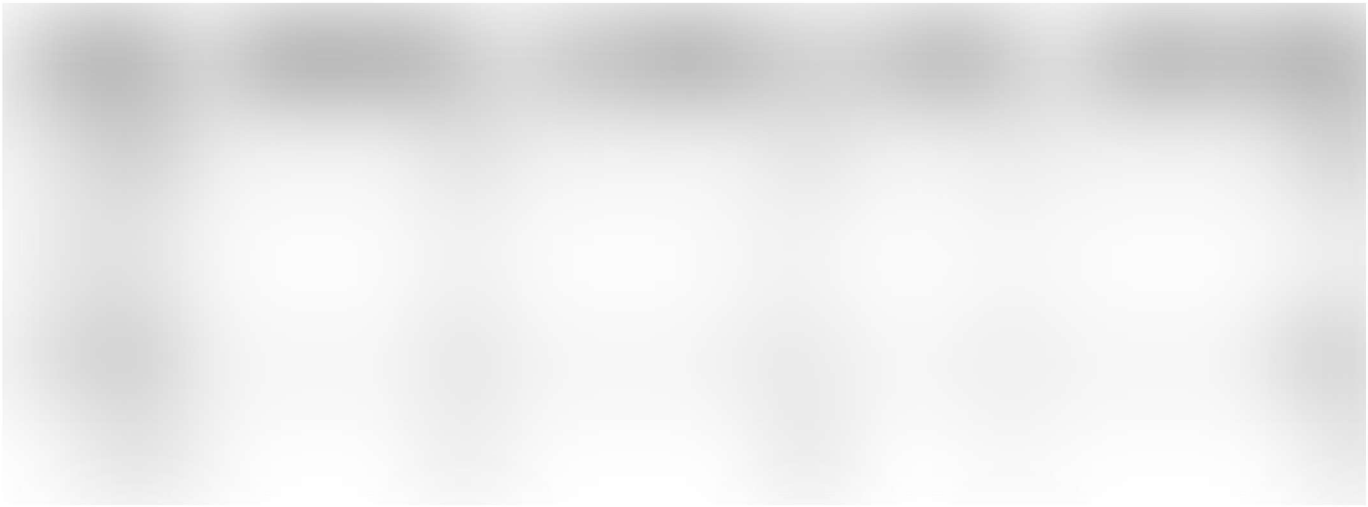
DIQA is an original proposal that focuses on solving some of the most concerning challenges of applying deep learning to image quality assessment (IQA). The advantages against other methodologies are:

- The model is not limited to work exclusively with Natural Scene Statistics (NSS) images [1].
- Prevents overfitting by splitting the training into two phases (1) feature learning and (2) mapping learned features to subjective scores.

Problem

The cost of generating datasets for IQA is high since it requires expert supervision. Therefore, the fundamental IQA benchmarks are comprised of solely a few thousands of records. The latter complicates the creation of deep learning models because they require large amounts of training samples to generalize.

For example, let's consider the most frequently used datasets to train and evaluate IQA methods Live, TID2008, TID2013, CSIQ. An overall summary of each dataset is contained in the next table:



The total amount of samples does not exceed 4,000 records for any of them.

Dataset

The IQA benchmarks only contain a limited amount of records that might not be enough to train a CNN. However, for this guide purpose, we are going to use the Live dataset. It is comprised of 29 reference images, and 5 different distortions with 5 severity levels each.

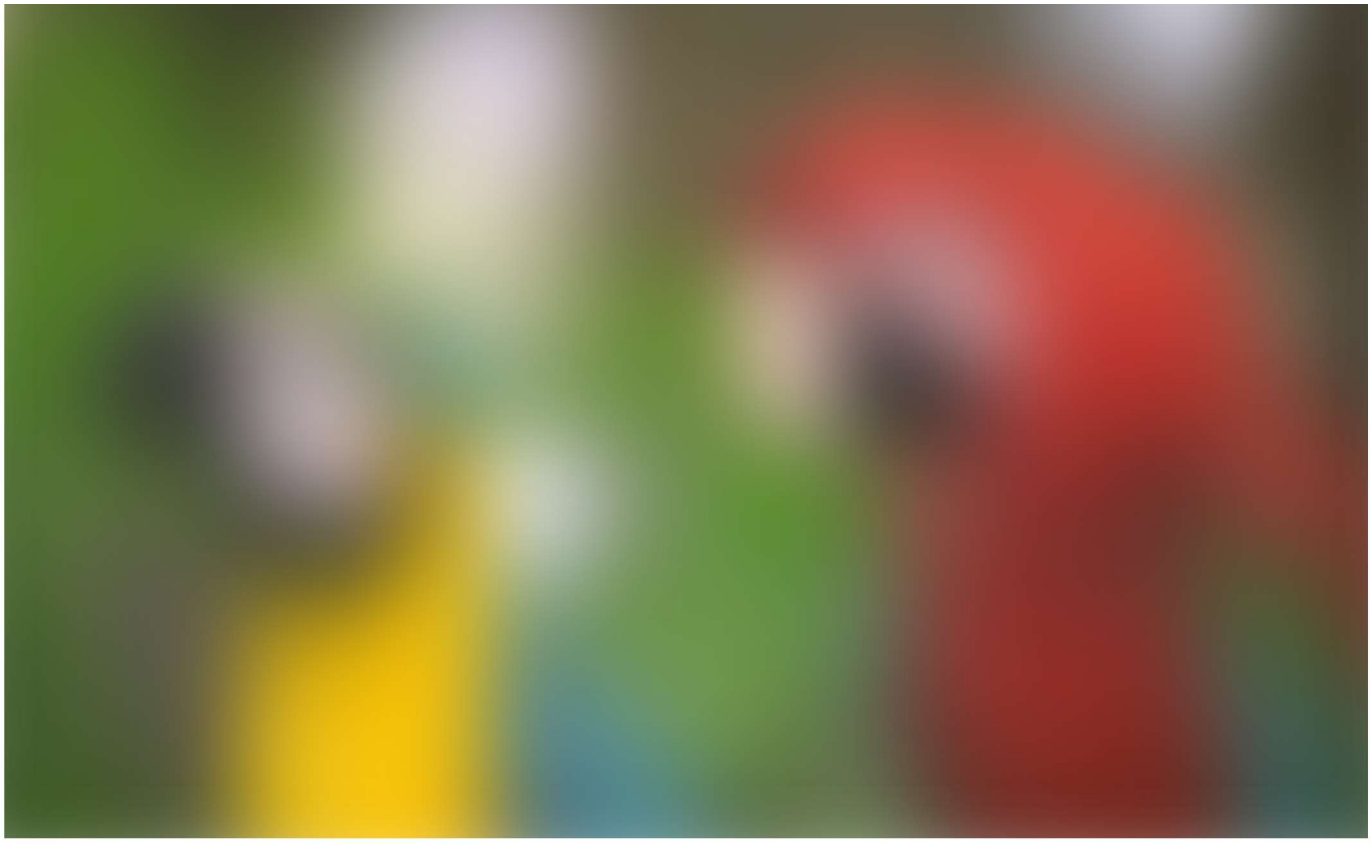


Fig 1. An example of a reference image in Live dataset.

The first task is to download and prepare the dataset. I have created a couple of TensorFlow dataset builders for image quality assessment and published them in the image-quality package. The builders are an interface defined by tensorflow-datasets.

Note: This process might take several minutes because of the size of the dataset (700 megabytes).

After downloading and preparing the data, turn the builder into a dataset, and shuffle it. Note that the batch is equal to 1. The reason is that each image has a different shape. Increasing the batch size will cause an error.

The output is a generator; therefore, accessing the samples using the bracket operator causes an error. There are two ways to access the images in the generator. The first way is to turn the generator into an iterator and extract a single sample using the *next* function.

The output is a dictionary that contains the tensor representation for the distorted image, the reference image, and the subjective score (dmos). Another way is to extract samples from the generator by taking them with a for loop:

Methodology

Image Normalization

The first step for DIQA is to pre-process the images. The image is converted into grayscale, and then a low-pass filter is applied. The low-pass filter is defined as:

where the low-frequency image is the result of the following algorithm:

1. Blur the grayscale image.
2. Downscale it by a factor of 1 / 4.

3. Upscale it back to the original size.

The main reasons for this normalization are (1) the Human Visual System (HVS) is not sensitive to changes in the low-frequency band, and (2) image distortions barely affect the low-frequency component of images.

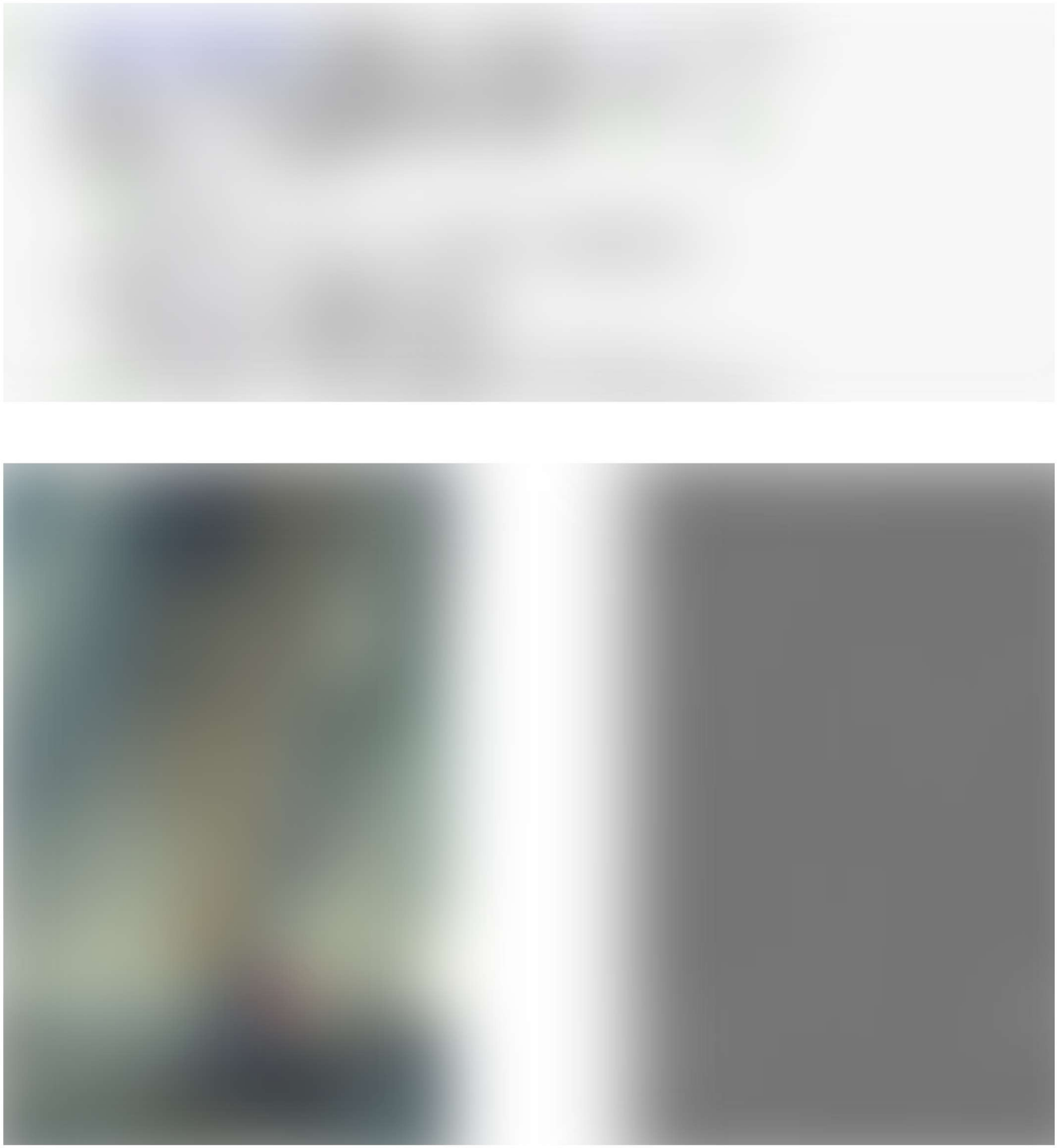


Fig 2. On the left, the original image. On the right, the image after applying the low-pass filter.

Objective Error Map

For the first model, objective errors are used as a proxy to take advantage of the effect of increasing data. The loss function is defined by the mean squared error between the predicted and ground-truth error maps.

and *err(·)* can be any error function. For this implementation, the authors recommend using

with $p=0.2$. The latter is to prevent that the values in the error map are small or close to zero.

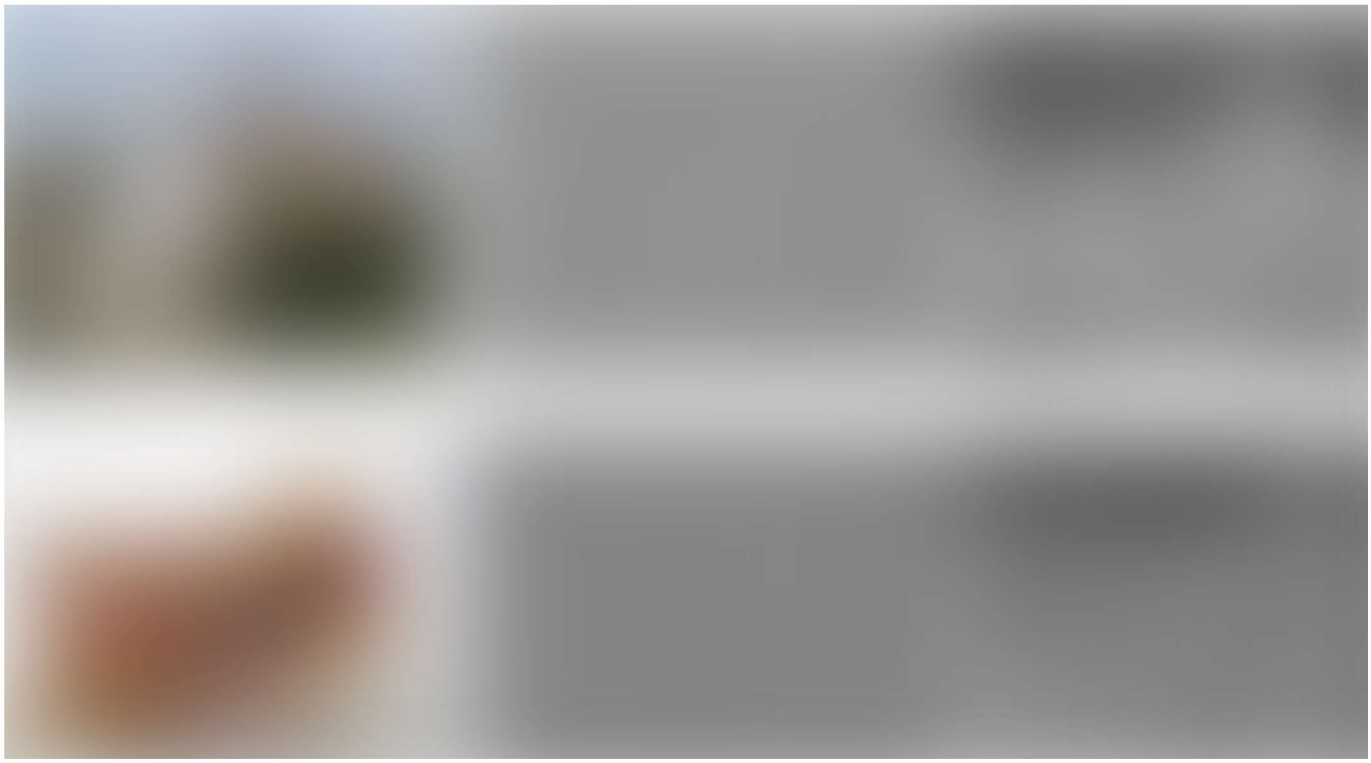


Fig 3. On the left, the original image. In the middle, the pre-processed image, and finally, the image representation of the error map.

Reliability Map

According to the authors, the model is likely to fail to predict images with homogeneous regions. To prevent it, they propose a reliability function. The assumption is that blurry areas have lower reliability than textured ones. The reliability function is defined as

where α controls the saturation property of the reliability map. The positive part of a sigmoid is used to assign sufficiently large values to pixels with low intensity.

The previous definition might directly affect the predicted score. Therefore, the average reliability map is used instead.

For the Tensorflow function, we just calculate the reliability map and divide it by its mean.

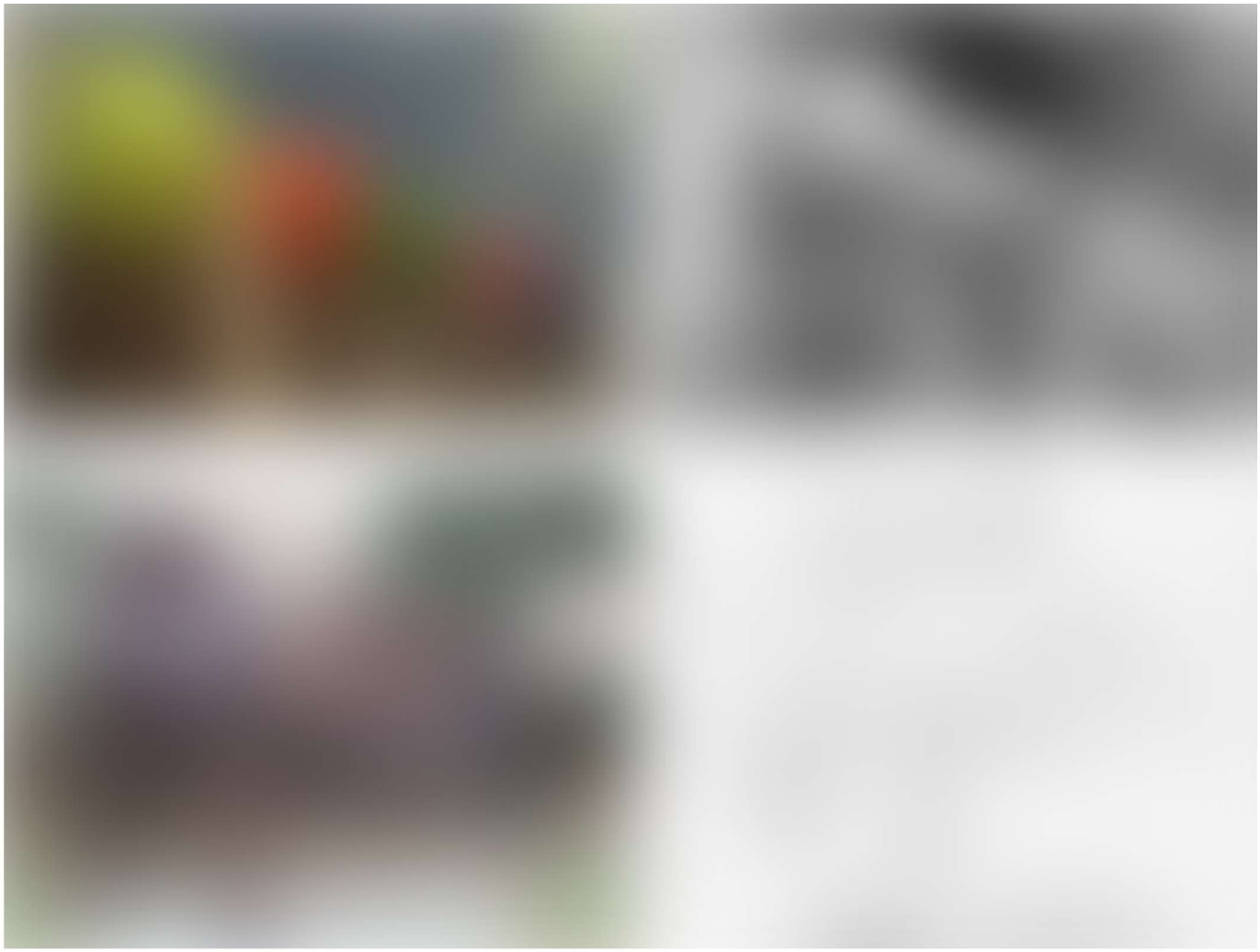


Fig 4. On the left, the original image, and on the right, its average reliability map.

Loss function

The loss function is defined as the mean square error of the product between the reliability map and the objective error map. The error is the difference between the predicted error map and the ground-truth error map.

The loss function requires to multiply the error by the reliability map; therefore, we cannot use the default loss implementation *tf.loss.MeanSquareError*.

After creating the custom loss, we need to tell TensorFlow how to differentiate it. The good thing is that we can take advantage of automatic differentiation using *tf.GradientTape*.

Optimizer

The authors suggested using a Nadam optimizer with a learning rate of *2e-4*.

Training

Objective Error Model

For the training phase, it is convenient to utilize the *tf.data* input pipelines to produce a much cleaner and readable code. The only requirement is to create the function to apply to the input.

Then, map the *tf.data.Dataset* to the *calculate_error_map* function.

Applying the transformation is executed in almost no time. The reason is that the processor is not performing any operation to the data yet, it happens on demand. This

concept is commonly called lazy-evaluation.

So far, the following components are implemented:

1. The generator that pre-processes the input and calculates the target.
2. The loss and gradient functions required for the custom training loop.
3. The optimizer function.

The only missing bits are the models' definition.

Fig 5. The architecture for the objective error map prediction. The red and blue arrows indicate the flows of the first and stage. Source: http://bit.ly/2Ldw4PZ

In the previous image, it is depicted how:

- The pre-processed image gets into the convolutional neural network (CNN).
- It is transformed by 8 convolutions with the Relu activation function and "same" padding. This is defined as $f(\cdot)$.
- The output of $f(\cdot)$ is processed by the last convolution with a linear activation function. This is defined as $g(\cdot)$.

For the custom training loop, it is necessary to:

1. Define a metric to measure the performance of the model.
2. Calculate the loss and the gradients.
3. Use the optimizer to update the weights.
4. Print accuracy.

Note: It would be a good idea to use the Spearman's rank-order correlation coefficient (SRCC) or Pearson's linear correlation coefficient (PLCC) as accuracy metrics.

Subjective Score Model

To create the subjective score model, let's use the output of f(·) to train a regressor.

Training a model with the fit method of *tf.keras.Model* expects a dataset that returns two arguments. The first one is the input, and the second one is the target.

Then, *fit* the subjective score model.

Prediction

Prediction using the already trained model is simple. Just use the *predict* method in the model.

Conclusion

In this article, we learned how to utilize the tf.data module to create easy to read and memory-efficient data pipelines. Also, we implemented the Deep CNN-Based Blind Image Quality Predictor (DIQA) model using the functional Keras API. The model was trained with a custom training loop that uses the auto differentiation feature from TensorFlow.

The next step is to find the hyperparameters that maximize the PLCC or SRCC accuracy metrics and evaluate the overall performance of the model against other methodologies.

Another idea is to use a much larger dataset to train the objective error map model and see the resulting overall performance.

Jupyter Notebook

ocampor/image-quality

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

bit.ly

Bibliography

[1] Kim, J., Nguyen, A. D., & Lee, S. (2019). Deep CNN-Based Blind Image Quality Predictor. IEEE Transactions on Neural Networks and Learning Systems. https://doi.org/10.1109/TNNLS.2018.2829819