BME646 and ECE60146: Homework 2

# Mykhailo Tsysin

# Jan 23 2023

In this homework, we will look more closely into how images are loaded and per-processed for subsequent use in a neural network For full .py solution, please look at corresponding file.

# Theory task:

The observed phenomena can be explained fairly simply. In the example on the slides, the maximum in the last image AND in the entire bath is 255. Also, the maximum occurs in the same image that we’re showing. The means that in both per image and per batch cases we divide by the said image by the same number 255, thus we get absolutely the same result.

# Programming tasks:

## Setting up conda environment:

After executing the command given in the homework assignment, we get the following environment setup. Note that CUDA tools are not installed since my commuter (MAC M1) lack a dedicated GPU. The homework can be completed otherwise though.

Text

Description automatically generated

## Apply transformations to the stop sign:

The available pictures of a stops sign cropped to 2048x2048 size are presented here:

Original:

A stop sign on the side of a street

Description automatically generated with medium confidence

Tilted:

A stop sign on the side of a road

Description automatically generated with medium confidence

First, here are some of the results of applying random affine and perspective transforms.

Affine:

Graphical user interface, application, calendar

Description automatically generated

Perspective:

Graphical user interface, application

Description automatically generated

The code for generating those instances is described here:

Text

Description automatically generated

For determining the exact transformation for the stop sign, we’ll use some analytical methods. Using Prof. Kak’s notes for ECE 661, we determine the following:

Suppose we have 4 points associated with the original image and a tilted image:

The transformation will be denoted as . The resulting transformation will look like:

By eliminating constants we can write the following equation:

Solving this, we find all the values for transformation matrix

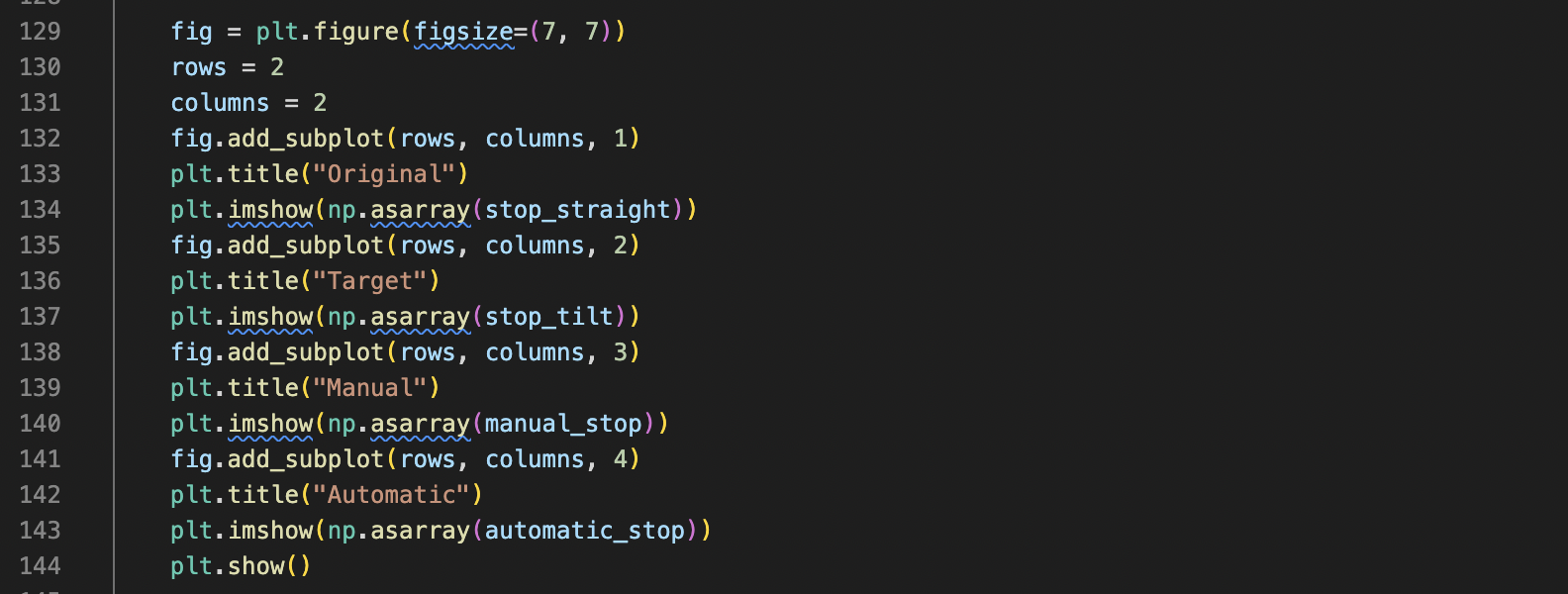
The code for extracting the needed parameters is presented:

Text

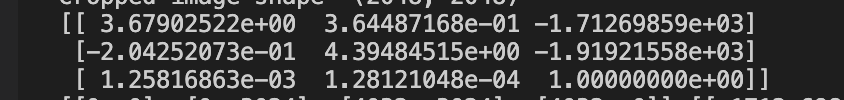
Description automatically generated

Text

Description automatically generated



H =

.

Before doing all of this I haven’t looked at the tvt.funcional.perspective syntax and didn’t know that it does everything for me from the original and transformed 4 points. Anyway, it’s nice to verify that my calculations are correct. Here are the original, target images with 2 transformed versions (automatic and “manual”):

Graphical user interface

Description automatically generated

We see that the stop sign is transformed exactly as it looks on the target image.

The histograms for the original, target, and resulting images were also calculated:

Text

Description automatically generated

The results were not very helpful mainly since the stop sign on the picture is relatively small, and backgrounds play significant role in color distributions. The numbers are smaller though. And ideally, we should see bigger difference in red channel. Unfortunately, there is a red car on the second picture, which spoils the results a bit.

Text

Description automatically generated

## Creating a dataset class:

The dataset class is created based on the given code with slight modifications. The raw variable is purely for disabling transform, can be safely ignored. Also, we don’t normalize image since it’s unclear how it will be used subsequently. It can be added easily:

Text

Description automatically generated

The output of presented driver code is:

Graphical user interface, text

Description automatically generated with medium confidence

Yes, the size is not standard, but that is what image “compressor” gave me.

The example of dataset output is shown below:

Graphical user interface, website

Description automatically generated

For transforms I chose gaussian blur (since it represents some focusing issues that might arise in a camera), affine transform (to be flexible with image position), and color jitter (to adapt for exposure and coloring issues). Also, it didn’t make sense to use other geometry transformations since perspective accounts for almost every possible scenario.

## Parallelization

We wrap the dataset into a dataloader and print one batch:

Text

Description automatically generated

The 4 generated images in a batch are:

Graphical user interface

Description automatically generated

Now, we’ll try to check the performance boost associated with using a dataloader. We’ll artificially change the list of images into a repeating sequence and reinitialize the dataloader. That will be a more representative test.

Text

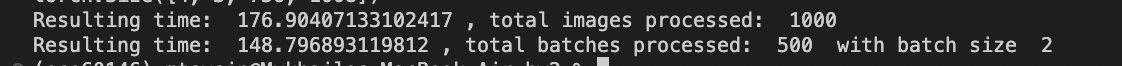
Description automatically generated

The resulting speed boost is marginal:

Batch = 4, workers = 4



Batch = 2, workers = 4



It’s interesting to note that the first time also changed. To make experiment clearer, I made a separate dataset consisting of 10 smaller 256 by 256 pictures:

Text

Description automatically generated

But the results remained unchanged. The data loader didn’t improve the performance:

|  |  |  |  |
| --- | --- | --- | --- |
| Workers/batches | 4 | 8 | 16 |
| 2 | 4.6s | 4.5s | 4.5s |
| 4 | 6.4s | 6.4s | 6.4s |

The original time was 4.2s. For the smaller images performance even became worse. Probably it can be attributed to some flaws in my system + threads taking too much time to set up.