PixelMind

September 6, 2023

1 PixelMind AI

[ ]:
```
#used to access files
import os

#used to generate random numbers or floats
import random

# used for array manipulation
import numpy as np

# used to serialize the data from folder for training/testing/validation
from glob import glob

 # used for final image comparison
from PIL import Image, ImageOps

import cv2
```

[ ]:
```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

[ ]:
```
## Creating a TensorFlow Dataset
```

[ ]:
```
IMAGE_SIZE = 4288, 2848
BATCH_SIZE = 8
MAX_TRAIN_IMAGES = 1400
MAX_VAL_IMAGES = 1490

# Split the dataset
train_low_light_images = sorted(glob("./drive/MyDrive/FinalDataset_Merged/
 ↪train_indoor/Raw/*"))[:MAX_TRAIN_IMAGES]
val_low_light_images = sorted(glob("./drive/MyDrive/FinalDataset_Merged/
 ↪train_indoor/Raw/*"))[MAX_TRAIN_IMAGES:MAX_VAL_IMAGES]
```

```
test_low_light_images = sorted(glob("./drive/MyDrive/FinalDataset_Merged/
↪train_indoor/Raw/*"))[MAX_VAL_IMAGES:]

def load_data(image_path):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=3) # Decode the image
    #image = tf.image.resize(images=image, size=[IMAGE_SIZE, IMAGE_SIZE])
    image = image / 2848 #normalize image
    return image


def data_generator(low_light_images):
    dataset = tf.data.Dataset.from_tensor_slices((low_light_images))
    dataset = dataset.map(load_data, num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
    return dataset

train_dataset = data_generator(train_low_light_images)
val_dataset = data_generator(val_low_light_images)

print("Train Dataset:", train_dataset)
print("Validation Dataset:", val_dataset)
```

[ ]:

```
"""## The Zero-DCE Framework

The goal of DCE-Net is to estimate a set of best-fitting light-enhancement␣
↪curves
(LE-curves) given an input image. The framework then maps all pixels of the␣
↪input's RGB
channels by applying the curves iteratively to obtain the final enhanced image.

### Understanding light-enhancement curves

A ligh-enhancement curve is a kind of curve that can map a low-light image
to its enhanced version automatically,
where the self-adaptive curve parameters are solely dependent on the input␣
↪image.
When designing such a curve, three objectives should be taken into account:

- Each pixel value of the enhanced image should be in the normalized range␣
↪`[0,1]`, in order to
avoid information loss induced by overflow truncation.
- It should be monotonous, to preserve the contrast between neighboring pixels.
- The shape of this curve should be as simple as possible,
and the curve should be differentiable to allow backpropagation.
```

```
The light-enhancement curve is separately applied to three RGB channels instead␣
↪of solely on the
illumination channel. The three-channel adjustment can better preserve the␣
↪inherent color and reduce
the risk of over-saturation.

![](https://li-chongyi.github.io/Zero-DCE_files/framework.png)

### DCE-Net

The DCE-Net is a lightweight deep neural network that learns the mapping␣
↪between an input
image and its best-fitting curve parameter maps. The input to the DCE-Net is a␣
↪low-light
image while the outputs are a set of pixel-wise curve parameter maps for␣
↪corresponding
higher-order curves. It is a plain CNN of seven convolutional layers with␣
↪symmetrical
concatenation. Each layer consists of 32 convolutional kernels of size 3×3 and␣
↪stride 1
followed by the ReLU activation function. The last convolutional layer is␣
↪followed by the
Tanh activation function, which produces 24 parameter maps for 8 iterations,␣
↪where each
iteration requires three curve parameter maps for the three channels.

![](https://i.imgur.com/HtIg34W.png)
"""
```

[ ]:
```
def build_dce_net():
    input_img = keras.Input(shape=[None, None, 3])
    conv1 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
    )(input_img)
    conv2 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
    )(conv1)
    conv3 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
    )(conv2)
    conv4 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
    )(conv3)
    int_con1 = layers.Concatenate(axis=-1)([conv4, conv3])
    conv5 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
```

```
    )(int_con1)
    int_con2 = layers.Concatenate(axis=-1)([conv5, conv2])
    conv6 = layers.Conv2D(
        32, (3, 3), strides=(1, 1), activation="relu", padding="same"
    )(int_con2)
    int_con3 = layers.Concatenate(axis=-1)([conv6, conv1])
    x_r = layers.Conv2D(24, (3, 3), strides=(1, 1), activation="tanh",␣
 ↪padding="same")(
        int_con3
    )
    return keras.Model(inputs=input_img, outputs=x_r)
```

[ ]:
```
"""## Loss functions

To enable zero-reference learning in DCE-Net, we use a set of differentiable
zero-reference losses that allow us to evaluate the quality of enhanced images.

### Color constancy loss

The *color constancy loss* is used to correct the potential color deviations in␣
 ↪the
enhanced image.
"""

def color_constancy_loss(x):
    mean_rgb = tf.reduce_mean(x, axis=(1, 2), keepdims=True)
    mr, mg, mb = mean_rgb[:, :, :, 0], mean_rgb[:, :, :, 1], mean_rgb[:, :, :,␣
 ↪2]
    d_rg = tf.square(mr - mg)
    d_rb = tf.square(mr - mb)
    d_gb = tf.square(mb - mg)
    return tf.sqrt(tf.square(d_rg) + tf.square(d_rb) + tf.square(d_gb))
```

[ ]:
```
"""### Exposure loss

To restrain under-/over-exposed regions, we use the *exposure control loss*.
It measures the distance between the average intensity value of a local region
and a preset well-exposedness level (set to `0.6`).
"""

def exposure_loss(x, mean_val=0.6):
    x = tf.reduce_mean(x, axis=3, keepdims=True)
    mean = tf.nn.avg_pool2d(x, ksize=16, strides=16, padding="VALID")
    return tf.reduce_mean(tf.square(mean - mean_val))
```

[ ]:
```
"""### Illumination smoothness loss
```

```
To preserve the monotonicity relations between neighboring pixels, the
*illumination smoothness loss* is added to each curve parameter map.
"""

def illumination_smoothness_loss(x):
    batch_size = tf.shape(x)[0]
    h_x = tf.shape(x)[1]
    w_x = tf.shape(x)[2]
    count_h = (tf.shape(x)[2] - 1) * tf.shape(x)[3]
    count_w = tf.shape(x)[2] * (tf.shape(x)[3] - 1)
    h_tv = tf.reduce_sum(tf.square((x[:, 1:, :, :] - x[:, : h_x - 1, :, :])))
    w_tv = tf.reduce_sum(tf.square((x[:, :, 1:, :] - x[:, :, : w_x - 1, :])))
    batch_size = tf.cast(batch_size, dtype=tf.float32)
    count_h = tf.cast(count_h, dtype=tf.float32)
    count_w = tf.cast(count_w, dtype=tf.float32)
    return 2 * (h_tv / count_h + w_tv / count_w) / batch_size
```

[ ]:
```
"""### Spatial consistency loss

The *spatial consistency loss* encourages spatial coherence of the enhanced␣
↪image by
preserving the contrast between neighboring regions across the input image and␣
↪its enhanced version.
"""

class SpatialConsistencyLoss(keras.losses.Loss):
    def __init__(self, **kwargs):
        super(SpatialConsistencyLoss, self).__init__(reduction="none")

        self.left_kernel = tf.constant(
            [[[[0, 0, 0]], [[-1, 1, 0]], [[0, 0, 0]]]], dtype=tf.float32
        )
        self.right_kernel = tf.constant(
            [[[[0, 0, 0]], [[0, 1, -1]], [[0, 0, 0]]]], dtype=tf.float32
        )
        self.up_kernel = tf.constant(
            [[[[0, -1, 0]], [[0, 1, 0]], [[0, 0, 0]]]], dtype=tf.float32
        )
        self.down_kernel = tf.constant(
            [[[[0, 0, 0]], [[0, 1, 0]], [[0, -1, 0]]]], dtype=tf.float32
        )

    def call(self, y_true, y_pred):

        original_mean = tf.reduce_mean(y_true, 3, keepdims=True)
        enhanced_mean = tf.reduce_mean(y_pred, 3, keepdims=True)
        original_pool = tf.nn.avg_pool2d(
```

```
            original_mean, ksize=4, strides=4, padding="VALID"
        )
        enhanced_pool = tf.nn.avg_pool2d(
            enhanced_mean, ksize=4, strides=4, padding="VALID"
        )

        d_original_left = tf.nn.conv2d(
            original_pool, self.left_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )
        d_original_right = tf.nn.conv2d(
            original_pool, self.right_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )
        d_original_up = tf.nn.conv2d(
            original_pool, self.up_kernel, strides=[1, 1, 1, 1], padding="SAME"
        )
        d_original_down = tf.nn.conv2d(
            original_pool, self.down_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )

        d_enhanced_left = tf.nn.conv2d(
            enhanced_pool, self.left_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )
        d_enhanced_right = tf.nn.conv2d(
            enhanced_pool, self.right_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )
        d_enhanced_up = tf.nn.conv2d(
            enhanced_pool, self.up_kernel, strides=[1, 1, 1, 1], padding="SAME"
        )
        d_enhanced_down = tf.nn.conv2d(
            enhanced_pool, self.down_kernel, strides=[1, 1, 1, 1],␣
↪padding="SAME"
        )

        d_left = tf.square(d_original_left - d_enhanced_left)
        d_right = tf.square(d_original_right - d_enhanced_right)
        d_up = tf.square(d_original_up - d_enhanced_up)
        d_down = tf.square(d_original_down - d_enhanced_down)
        return d_left + d_right + d_up + d_down
```

[ ]:

```
"""### Deep curve estimation model

We implement the Zero-DCE framework as a Keras subclassed model.
```

```
"""

class ZeroDCE(keras.Model):
    def __init__(self, **kwargs):
        super(ZeroDCE, self).__init__(**kwargs)
        self.dce_model = build_dce_net()

    def compile(self, learning_rate, **kwargs):
        super(ZeroDCE, self).compile(**kwargs)
        self.optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
        self.spatial_constancy_loss = SpatialConsistencyLoss(reduction="none")

    def get_enhanced_image(self, data, output):
        r1 = output[:, :, :, :3]
        r2 = output[:, :, :, 3:6]
        r3 = output[:, :, :, 6:9]
        r4 = output[:, :, :, 9:12]
        r5 = output[:, :, :, 12:15]
        r6 = output[:, :, :, 15:18]
        r7 = output[:, :, :, 18:21]
        r8 = output[:, :, :, 21:24]
        x = data + r1 * (tf.square(data) - data)
        x = x + r2 * (tf.square(x) - x)
        x = x + r3 * (tf.square(x) - x)
        enhanced_image = x + r4 * (tf.square(x) - x)
        x = enhanced_image + r5 * (tf.square(enhanced_image) - enhanced_image)
        x = x + r6 * (tf.square(x) - x)
        x = x + r7 * (tf.square(x) - x)
        enhanced_image = x + r8 * (tf.square(x) - x)
        return enhanced_image

    def call(self, data):
        dce_net_output = self.dce_model(data)
        return self.get_enhanced_image(data, dce_net_output)

    def compute_losses(self, data, output):
        enhanced_image = self.get_enhanced_image(data, output)
        loss_illumination = 200 * illumination_smoothness_loss(output)
        loss_spatial_constancy = tf.reduce_mean(
            self.spatial_constancy_loss(enhanced_image, data)
        )
        loss_color_constancy = 5 * tf.
↪reduce_mean(color_constancy_loss(enhanced_image))
        loss_exposure = 10 * tf.reduce_mean(exposure_loss(enhanced_image))
        total_loss = (
            loss_illumination
            + loss_spatial_constancy
```

```
            + loss_color_constancy
            + loss_exposure
        )
        return {
            "total_loss": total_loss,
            "illumination_smoothness_loss": loss_illumination,
            "spatial_constancy_loss": loss_spatial_constancy,
            "color_constancy_loss": loss_color_constancy,
            "exposure_loss": loss_exposure,
        }

    def train_step(self, data):
        with tf.GradientTape() as tape:
            output = self.dce_model(data)
            losses = self.compute_losses(data, output)
        gradients = tape.gradient(
            losses["total_loss"], self.dce_model.trainable_weights
        )
        self.optimizer.apply_gradients(zip(gradients, self.dce_model.
↪trainable_weights))
        return losses

    def test_step(self, data):
        output = self.dce_model(data)
        return self.compute_losses(data, output)

    def save_weights(self, filepath, overwrite=True, save_format=None,␣
↪options=None):
        """While saving the weights, we simply save the weights of the␣
↪DCE-Net"""
        self.dce_model.save_weights(
            filepath, overwrite=overwrite, save_format=save_format,␣
↪options=options
        )

    def load_weights(self, filepath, by_name=False, skip_mismatch=False,␣
↪options=None):
        """While loading the weights, we simply load the weights of the␣
↪DCE-Net"""
        self.dce_model.load_weights(
            filepath=filepath,
            by_name=by_name,
            skip_mismatch=skip_mismatch,
            options=options,
        )
```

```
[ ]: """## Training"""

zero_dce_model = ZeroDCE()
zero_dce_model.compile(learning_rate=0.4)
history = zero_dce_model.fit(train_dataset, validation_data=val_dataset,␣
 ↪epochs=200)


def plot_result(item):
    plt.plot(history.history[item], label=item)
    plt.plot(history.history["val_" + item], label="val_" + item)
    plt.xlabel("Epochs")
    plt.ylabel(item)
    plt.title("Train and Validation {} Over Epochs".format(item), fontsize=14)
    plt.legend()
    plt.grid()
    plt.show()
```

```
[ ]: plot_result("total_loss")
plot_result("illumination_smoothness_loss")
plot_result("spatial_constancy_loss")
plot_result("color_constancy_loss")
plot_result("exposure_loss")
```

```
[ ]: """## Inference"""

def plot_results(images, titles, figure_size=(12, 12)):
    fig = plt.figure(figsize=figure_size)
    for i in range(len(images)):
        fig.add_subplot(1, len(images), i + 1).set_title(titles[i])
        _ = plt.imshow(images[i])
        plt.axis("off")
    plt.show()


def infer(original_image):
    image = keras.preprocessing.image.img_to_array(original_image)
    image = image.astype("float32") / 255.0
    image = np.expand_dims(image, axis=0)
    output_image = zero_dce_model(image)
    output_image = tf.cast((output_image[0, :, :, :] * 255), dtype=np.uint8)
    output_image = Image.fromarray(output_image.numpy())
    return output_image
```

```
[ ]: """### Inference on test images and visualize results using W&B Tables

"""
```

```python
wandb.init(project="low_light_zero_DCE", job_type="predictions")

table = wandb.Table(columns=["Original", "PIL Autocontrast", "Enhanced"])
for val_image_file in test_low_light_images:
    original_image = Image.open(val_image_file)
    enhanced_image = infer(original_image)
    table.add_data(
        wandb.Image(np.array(original_image)),
        wandb.Image(np.array(ImageOps.autocontrast(original_image))),
        wandb.Image(np.array(enhanced_image))
    )

wandb.log({"Inference Table": table})

wandb.finish()
```

[ ]:
```python
def get_size_format(b, factor=1024, suffix="B"):
```