Vision Datasets

INTRODUCTION

Various datasets for computer vision projects have been compiled globally. We have gathered as much data as possible, focusing on content and task relevance. Creating a custom dataset is not straightforward; one significant challenge is the volume of records required. It is unrealistic for individuals or organizations to provide extensive datasets comprising images, labels, and videos. Assuming a large vision dataset could be assembled, what ensures the quality of the images and videos? In this discussion, we outline several checkpoints to ensure our dataset remains pure and efficient for use."

Datasets common sources

1. Kaggle
2. Torchvision-Dataset
3. Visualdata
4. Roboflow
5. HuggingfaceImageNet
6. Vision-datasets
7. UCI
8. Google Dataset Search
9. Amazon Dataset
10. Paper with code

Famous Datasets

a. Caltech 101
b. Caltech 256
c. COYO-700M
d. SIFT10M Dataset
e. LabelMe
f. PASCAL VOC Dataset

g. [CIFAR-10]
h. [CIFAR-100]
i. [CINIC-10]
j. [Fashion MNIST]
k. [notMNIST]
l. [Linnaeus 5]
m. [SVHN]
n. [CelebA]
o. [CitySpaces]
p. [ShapeNet]
q. [nuScenes]
r. [ScanNet]
s. [Stanford Cars]
t. [DTD]
u. [BSD]
v. [Kinetics 400]
w. [DomainNet]
x. [Food 101]
y. [CheXpert]
z. [iNaturalist]
aa. …

To advance in identifying comprehensive datasets, I developed a script utilizing web scraping techniques to compile a list of dataset names relevant to various tasks. This approach leverages the fact that many of these datasets are readily available through the pre-installed PyTorch library.

```
def table_dataset(url):

   response = requests.get(url)

   soup = BeautifulSoup(response.text, 'html.parser')
```

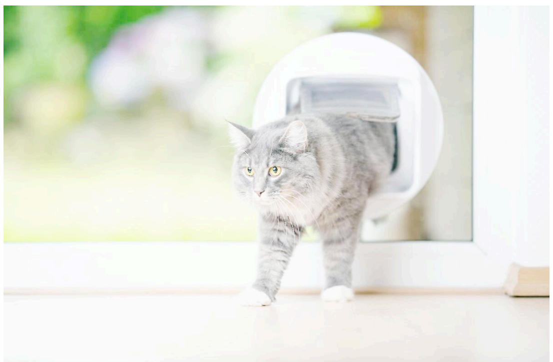
```
    # Find all h3 tags and extract their text content
    h3_tags = [h3.text.strip()[:-1] for h3 in soup.find_all('h3')]
    # Assuming h3_tags is defined somewhere above this snippet
    table_dataset = {}
    for tag in h3_tags:
        section_soup = soup.find_all('section', id=tag.lower().replace(' ',
'-'))
        section_soup = BeautifulSoup(str(section_soup), 'html.parser')
        # Initialize an empty list for each section if it doesn't already
exist
        if tag not in table_dataset:
            table_dataset[tag] = []
        # Append the text of each <span> element to the corresponding
section's list
        for element in section_soup.find_all('span'):
            table_dataset[tag].append(element.get_text())
    return table_dataset
# Example usage
url = 'https://pytorch.org/vision/main/datasets.html'
print(table_dataset(url))
```

PROCEDURE

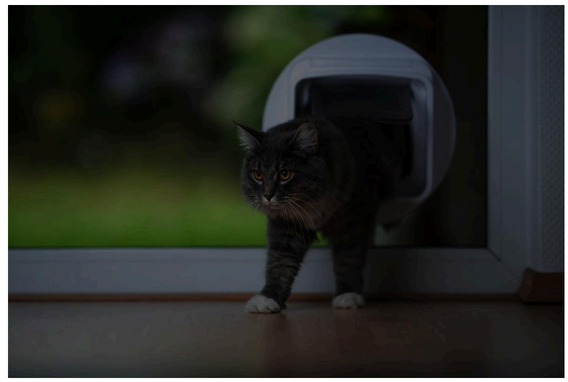
A valid dataset must follow some rules in spite of concept and size for a desired project.

Brightness

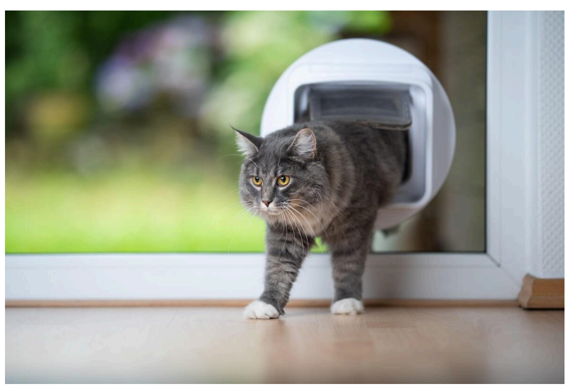
First most of the images should not be too bright or too dark



Too bright



Too dark



Right brightness

Brightness detection

```python
def normalize_value(value, min_val=20, max_val=250):
    """Normalize a value to the range [0, 1]."""
    return (value - min_val) / (max_val - min_val)
# Assuming you've already loaded the CIFAR-10 dataset as shown previously
train_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=None)

# Function to calculate the brightness of an image
def calculate_brightness(image_array):
    """
    Calculate the perceived brightness of an image represented as a NumPy array.
    """
    # Calculate the RMS of the RGB channels
    r, g, b = image_array[:,:,0], image_array[:,:,1], image_array[:,:,2]
    rms_rgb = np.sqrt((r**2).mean() + (g**2).mean() + (b**2).mean())

    # Calculate the perceived brightness
    brightness = math.sqrt(0.241 * (rms_rgb**2))
    brightness = normalize_value(brightness, min_val=20, max_val=250)
    return brightness

# Calculate brightness for each image in the training dataset
for i, (image, label) in enumerate(train_dataset):
    # Convert PIL Image to numpy array and then to float for accurate calculations
    image_np = np.array(image).astype(np.float32)
    brightness = calculate_brightness(image_np)
```

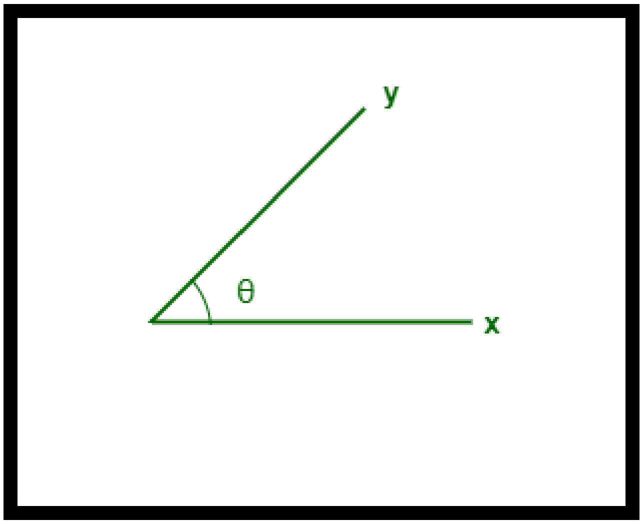
Similarity detection

A valid dataset should not include images that are similar to each other or appear identical, encompassing both duplicates and near-duplicates.

Near duplicate

Exact duplicate

In programming, we often assume that all data can be represented as tensors, possessing magnitude and direction within their domain, governed by tensor behavior. Cosine relationships allow us to quantify the closeness of two tensors. Therefore, calculating the cosine similarity between each element of the dataset enables us to numerically identify similarities among them.

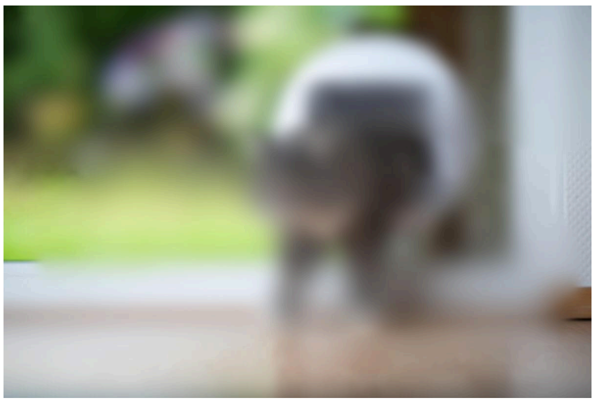


$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

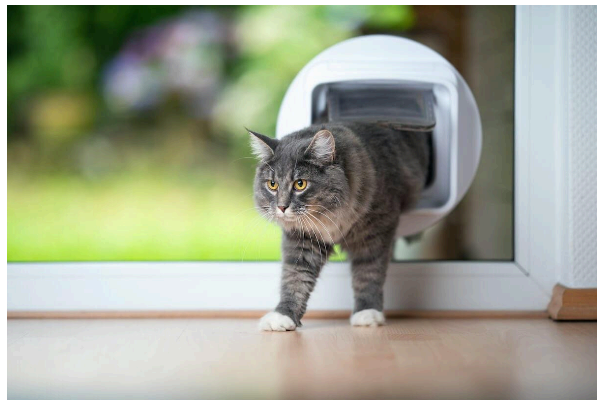
```
train_tensors = []
for i in range(len(train_dataset)):
 train_tensors.append(train_dataset[i][0])


flatten_train_dataset = []
for image in train_tensors:
 image = np.array(image)
 image = image.flatten()
 flatten_train_dataset.append(image)


from sklearn.metrics.pairwise import cosine_similarity
cosine_similarity(flatten_train_dataset)
```

Blurry

All images within a valid dataset should be sufficiently clear; however, some may appear blurry due to unintended movements. To construct an efficient dataset, it is imperative to include high-quality and crisp images.



Blurry



Clear

In my view, we should initially focus on the tensor aspect without considering color. By applying a Fourier transformation to the image, we can examine the distribution of low and high frequencies. This analysis allows us to determine if the frequency density is evenly distributed, which would indicate potential blurring issues.

```
def detect_blur_fft(img):

    # Convert image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
```

```
# Compute FFT of the original and blurred images

dft_gray = cv2.dft(np.float32(gray), flags=cv2.DFT_COMPLEX_OUTPUT)

dft_blurred = cv2.dft(np.float32(blurred),
flags=cv2.DFT_COMPLEX_OUTPUT)

# Split the complex spectrum into magnitude and phase components

magnitude_spectrum_gray = 20 * np.log(cv2.magnitude(dft_gray[:,:,0],
dft_gray[:,:,1]))

magnitude_spectrum_blurred = 20 *
np.log(cv2.magnitude(dft_blurred[:,:,0], dft_blurred[:,:,1]))

# Subtract the magnitude spectra of the blurred and original images

difference = cv2.subtract(magnitude_spectrum_blurred,
magnitude_spectrum_gray)

# Threshold the difference to get binary mask

_, thresholded = cv2.threshold(difference, 30, 255, cv2.THRESH_BINARY)

# Find contours in the thresholded image

contours, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

# Draw contours on the original image

cv2.drawContours(img, contours, -1, (0, 255, 0), 3)

# Display the result

cv2.imshow('Detected Blur', img)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Abundance of odd image size

The only thing this needs is a plot of distribution.

CONCLUSION

To assess the validity of our dataset, we examine the distribution of each specified quantity. By plotting these distributions, we can determine if they adhere to a normal distribution. A normal distribution suggests our dataset is acceptable. However, if the distribution deviates from normality, we must question the dataset's validity.

It's important to note that occasional outliers do not necessarily invalidate the entire dataset. These anomalies can be addressed at a later stage after further analysis.