Image Augmentation

Image augmentation **generates random images based on existing training data to improve the generalization ability of models**.

1. Random Transformation

- An helper function to apply transform the image by randomly choosing these options.
    1. flip_left_right
    2. flip_up_down
    3. rot90
- Inbuilt image transformation functions are available in TensorFlow.

```
def random_transform(input_image):
    """
    Takes in an input image and creates a new image out of it using a
        random transformation.
    Args:
        input_image: A PIL image object.

    Returns:
        output_image: A PIL image object, the augmented image.
    """

    # convert PIL image to numpy array.
    input_array = img_to_array(input_image)

    # Randomly choose transformation.
    transformation = np.random.choice([
        'flip_left_right',
        'flip_up_down',
        'rot90'
    ])

    if transformation == 'flip_left_right':
        print("Flipping the image left to right.")
        output_array = tf.image.flip_left_right(input_array)
    elif transformation == 'flip_up_down':
        print("Flipping the image up to down.")
        output_array = tf.image.flip_up_down(input_array)
    elif transformation == 'rot90':
        k = np.random.randint(1, 4)  # Randomly choose rotation angle (1, 2, or 3)
        print("Rotating the image")
        output_array = tf.image.rot90(input_array, k)

    # convert numpy array back to PIL Image.
    output_image = array_to_img(output_array)

    return output_image
```

2. Divide by Classes

Dividing the images by classes can help us go through each classes and augment the data according to class which has maximum number of datapoints / images.

```
def divide_data_by_class(input_images, image_labels):
    '''
    Divides the input images and labels into subsets based on their corresponding class labels.

    Args:
        input_images: A list of input images to be divided based on class
            labels. Each input image must be a PIL image.
        image_labels: A list that contains corresponding labels for each
            image in input_images. Each label is a string.

    Returns:
        classwise_images: A list of lists, where each sublist contains the
            input images corresponding to a unique class label.
        classwise_labels: A list of lists, where each sublist contains the
            labels corresponding to the input images in classwise_images list.
    '''
```

```
    # Initialize dictionaries to store images and labels by class
    classwise_images = {}
    classwise_labels = {}

    # Iterate over input images and labels
    for image, label in zip(input_images, image_labels):
        # Check if class label already exists in dictionaries
        if label not in classwise_images:
            classwise_images[label] = []
            classwise_labels[label] = []

        # Add image and label to respective class
        classwise_images[label].append(image)
        classwise_labels[label].append(label)

    # Convert dictionaries to lists of lists
    classwise_images = list(classwise_images.values())
    classwise_labels = list(classwise_labels.values())

    return classwise_images, classwise_labels
```

3. Implementing Augmentation.

By help of the functions above the augmentation process will be made easy.

The steps of algorithm are:

1. Get class wise images and labels. `divide_data_by_class`
2. Find the class with max count .
3. Calculate target size for each class after augmentation
   a. if the `data_size_factor` is ***2x*** and `max_class_size` is 4 then $4 * 2 = 8$
4. Loop through the `classwise_images` and `classwise_labels` to augment the data.
   a. Store the augmented data in the lists.

```
def augment_data(input_images, image_labels, data_size_factor):
    '''
    Augments the training data by randomly applying data augmentation techniques.

    Args:
        input_images: A list of input images.
        image_labels: A list of labels for the input images.
        data_size_factor: A scaling factor for the size of the augmented data.
            This will be used to calculate the final size of each class by
            multiplying data_size_factor with the size of the largest class and
            then rounding to the nearest integer.

    Returns:
        new_images: The augmented images
        new_labels: The labels corresponding to the new images
    '''

    # Divide input data by class
    classwise_images, classwise_labels = divide_data_by_class(input_images, image_labels)

    # Find the size of the largest class
    max_class_size = max(len(images) for images in classwise_images)

    # Calculate target size for each class after augmentation
    target_size = round(max_class_size * data_size_factor)

    # Augment data for each class
    new_images = []
    new_labels = []
    for images, labels in zip(classwise_images, classwise_labels):
        num_images_to_augment = target_size - len(images)
        if num_images_to_augment > 0:
            new_images.extend(images)
            new_labels.extend(labels)
            for _ in range(num_images_to_augment):
                # Randomly choose an image to augment
                image_to_augment = random.choice(images)
                # Apply random transformation
                augmented_image = random_transform(image_to_augment)
```

```
                # Add augmented image and label
                new_images.append(augmented_image)
                new_labels.append(labels[0])  # Assume all images in the class have the same label

    return new_images, new_labels
```

```
# Augment your training data using the "augment_data()" function
X_train, y_train = augment_data(X_train, y_train, 6)
X_test, y_test = augment_data(X_test, y_test, 12)
```