Image Data Augmentation on the Pet Data

Import Libraries

In [1]:
```
%%sh
pip install -q albumentations
pip install -q --upgrade wandb
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system pack
age manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behavio
ur is the source of the following dependency conflicts.
allennlp 2.7.0 requires wandb<0.13.0,>=0.10.0, but you have wandb 0.15.12 which is incompatible.
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system pack
age manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv
```

In [2]:
```
import gc
import os
import glob
import sys
import cv2
import imageio
import joblib
import math
import random
import wandb
import math

import numpy as np
import pandas as pd

from scipy.stats import kstest

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.image as mpimg

from PIL import Image

from statsmodels.graphics.gofplots import qqplot

plt.rcParams.update({'font.size': 18})
plt.style.use('fivethirtyeight')

import seaborn as sns
import matplotlib

from termcolor import colored

from multiprocessing import cpu_count
from tqdm.notebook import tqdm
from sklearn.model_selection import StratifiedKFold
from scipy.stats import pearsonr

import torch
import transformers
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset

import torchvision
import torchvision.transforms as transforms

from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import r2_score, mean_squared_error

from albumentations import (
    HorizontalFlip, VerticalFlip, IAAPerspective, ShiftScaleRotate, CLAHE, RandomRotate90,
    Transpose, ShiftScaleRotate, Blur, OpticalDistortion, GridDistortion, HueSaturationValue,
    IAAAdditiveGaussianNoise, GaussNoise, MotionBlur, MedianBlur, IAAPiecewiseAffine, RandomResizedCrop,
    IAASharpen, IAAEmboss, RandomBrightnessContrast, Flip, OneOf, Compose, Normalize, Cutout, CoarseDropout, ShiftScale
)
from albumentations.pytorch import ToTensorV2

# Core layers
from keras.layers \
    import Activation, Dropout, Flatten, Dense, Input, LeakyReLU

# Normalization layers
from keras.layers import BatchNormalization

# Merge layers
from keras.layers import concatenate, multiply

# Embedding Layers
from keras.layers import Embedding

# Keras models
from keras.models import Model, Sequential

# Keras optimizers
```

```
from tensorflow.keras.optimizers import Adam, RMSprop, SGD

import warnings
warnings.simplefilter('ignore')

# Activate pandas progress apply bar
tqdm.pandas()
```

In [3]:
```
# Wandb Login
import wandb
wandb.login()
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
```

```
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
```

Out[3]:
```
True
```

Global Config

We shall declare all the required configurations in the `config` class so that it comes in handy throughout the code.

In [4]:
```
class config:
    DIRECTORY_PATH = "../input/petfinder-pawpularity-score"
    TRAIN_FOLDER_PATH = DIRECTORY_PATH + "/train"
    TRAIN_CSV_PATH = DIRECTORY_PATH + "/train.csv"
    TEST_CSV_PATH = DIRECTORY_PATH + "/test.csv"
    SEED = 42
```

In [5]:
```
# wandb config
WANDB_CONFIG = {
     'competition': 'PetFinder',
              '_wandb_kernel': 'neuracort'
    }
```

We set the seed to a standard value.

In [6]:
```
def set_seed(seed=config.SEED):
    random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)
    np.random.seed(seed)

    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed()
```

Weights and Biases

Weights & Biases is the machine learning platform for developers to build better models faster.

You can use W&B's lightweight, interoperable tools to

- quickly track experiments,
- version and iterate on datasets,
- evaluate model performance,
- reproduce models,
- visualize results and spot regressions,
- and share findings with colleagues.

Set up W&B in 5 minutes, then quickly iterate on your machine learning pipeline with the confidence that your datasets and models are tracked and versioned in a reliable system of record.

Load Datasets

We define a function `return_filepath` which returns the file_path of an image which we require.

In [7]:
```
def return_filpath(name, folder=config.TRAIN_FOLDER_PATH):
    path = os.path.join(folder, f'{name}.jpg')
    return path
```

In [8]:
```
train = pd.read_csv(config.TRAIN_CSV_PATH)
train['image_path'] = train['Id'].apply(lambda x: return_filpath(x))
```

Now we have the dataframe ready and we can move to the augmentations part.

In [9]:
```
train.head()
```

Out[9]:

	Id	Subject Focus	Eyes	Face	Near	Action	Accessory	Group	Collage	Human	Occlusion	Info	Blur	Pawpul
0	0007de18844b0dbbb5e1f607da0606e0	0	1	1	1	0	0	1	0	0	0	0	0	
1	0009c66b9439883ba2750fb825e1d7db	0	1	1	0	0	0	0	0	0	0	0	0	
2	0013fd999caf9a3efe1352ca1b0d937e	0	1	1	1	0	0	0	0	1	1	0	0	
3	0018df346ac9c1d8413cfcc888ca8246	0	1	1	1	0	0	0	0	0	0	0	0	
4	001dc955e10590d3ca4673f034feeef2	0	0	0	1	0	0	1	0	0	0	0	0	

What is Data Augmentation?

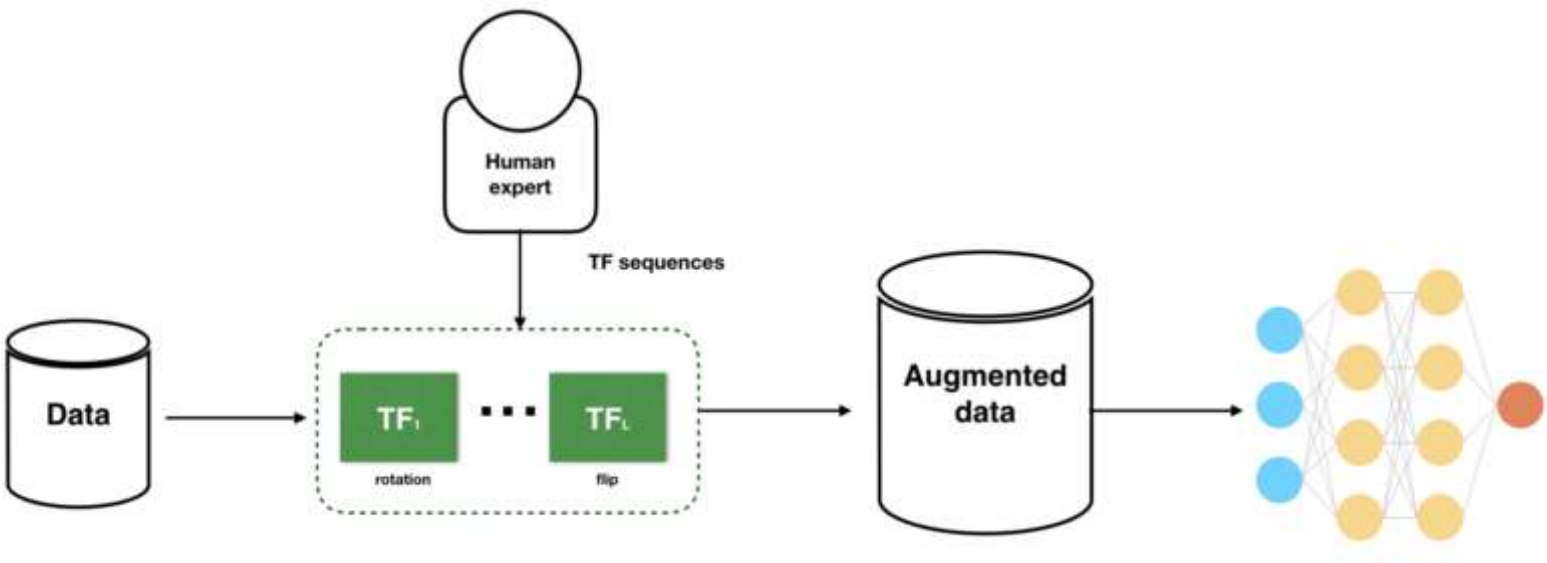
Data Augmentation is a technique used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.

Data augmentation is useful to improve performance and outcomes of machine learning models by forming new and different examples to train datasets. If dataset in a machine learning model is rich and sufficient, the model performs better and more accurate.

For machine learning models, collecting and labeling of data can be exhausting and costly processes. Transformations in datasets by using data augmentation techniques allow companies to reduce these operational costs.

How does Data Augmentation Work?

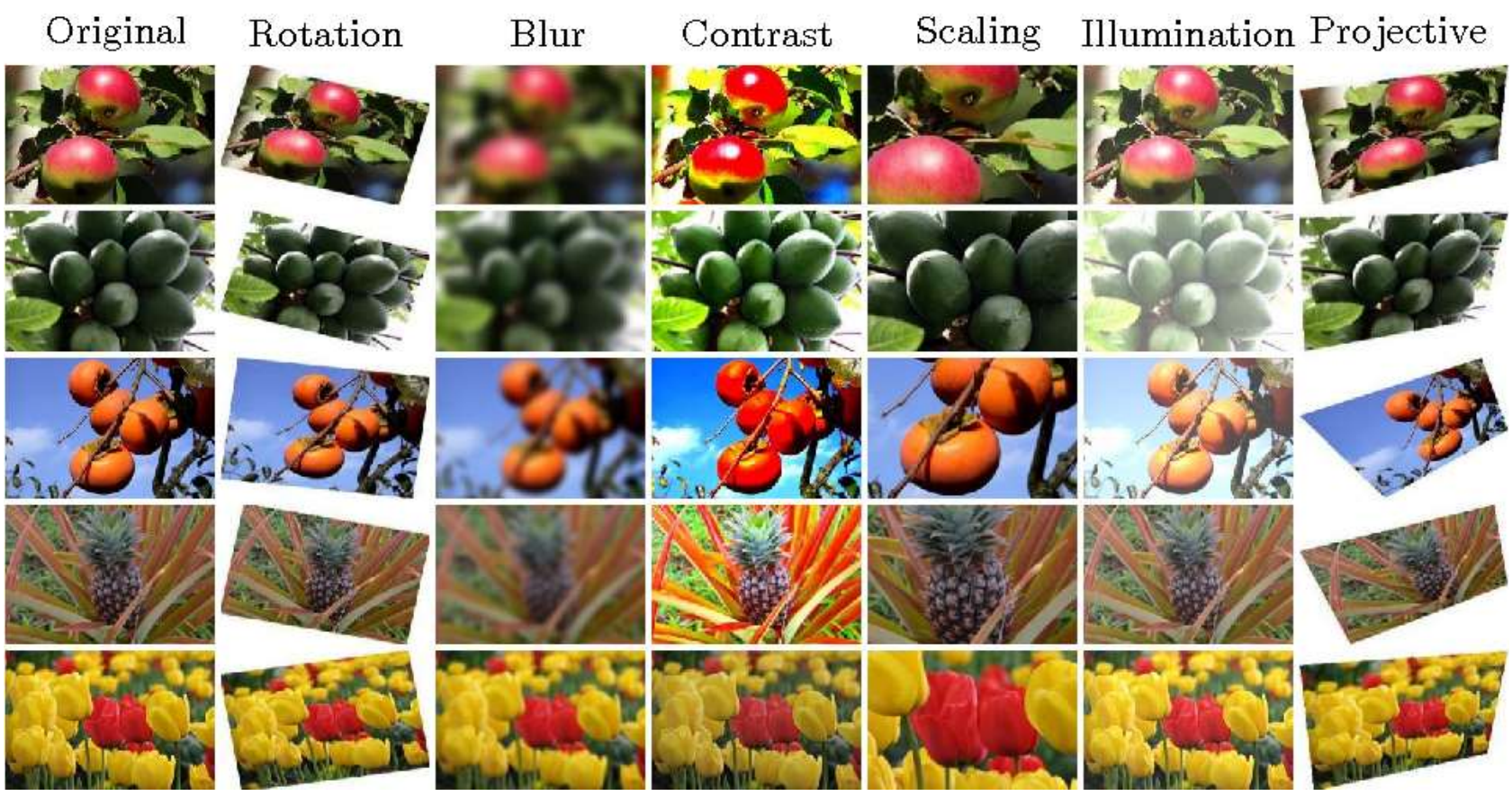
Computer vision applications use common data augmentation methods for training data. There are classic and advanced techniques in data augmentation for image recognition and natural language processing.



For data augmentation, making simple alterations on visual data is popular. In addition, generative adversarial networks (GANs) are used to create new synthetic data.

1. Classical Methods **Classic image processing activities for data augmentation are:**

- Padding
- Random rotating
- Re-scaling,
- Vertical and horizontal flipping
- Translation (image is moved along X, Y direction)
- Cropping
- Zooming
- Darkening & brightening/color modification
- Grayscaling
- Changing contrast
- Adding noise
- Random erasing



1. Advanced Methods **Advanced models for data augmentation are:**

- Adversarial training/Adversarial machine learning:
  It generates adversarial examples which disrupt a machine learning model and injects them into dataset to train.
- Generative adversarial networks (GANs):
  GAN algorithms can learn patterns from input datasets and automatically create new examples which resemble into training data.
- Neural style transfer:
  Neural style transfer models can blend content image and style image and separate style from content.
- Reinforcement learning:
  Reinforcement learning models train software agents to reach attain their goals and make decisions in a virtual environment.

Popular open source python packages for data augmentation in computer vision are Keras ImageDataGenerator, Skimage and OpenCV.

Basic Image Augmentation

In this section we will see what are the different **color spaces** in which we can transform the images. Based on your target you can select suitable color spaces and train model on that dataset.

I will demonstrate 9 different variations of color spaces namely:

1. Black and White
2. Ben Graham: Greyscale + Gaussian Blur
3. Hue, Saturation, Brightness
4. LUV Color Space
5. Alpha Channel
6. XYZ Color Space
7. Luma Chroma
8. CIE Lab
9. YUV Color Space

We shall define a simple function which will plot the augmented images for us.

In [10]:
```
def plot_augments(title, color_space):
    """
    Function which plots a 2*6 plot of images of specified color space.

    params: title(str)  - Title of the Plot
            color_space - color_space to which we convert the images
    """

    # Define subplots
    fig, axes = plt.subplots(nrows=2, ncols=6, figsize=(16,6))
    plt.suptitle(title, fontsize = 16)

    # Loop through images
    for i in range(0, 2*6):
        image = cv2.imread(train['image_path'][i])

        # The function converts an input image from one color space to another.
        image = cv2.cvtColor(image, color_space)
        image = cv2.resize(image, (200,200))

        x = i // 6
        y = i % 6
```

```
        axes[x, y].imshow(image, cmap=plt.cm.bone)
        axes[x, y].axis('off')
```

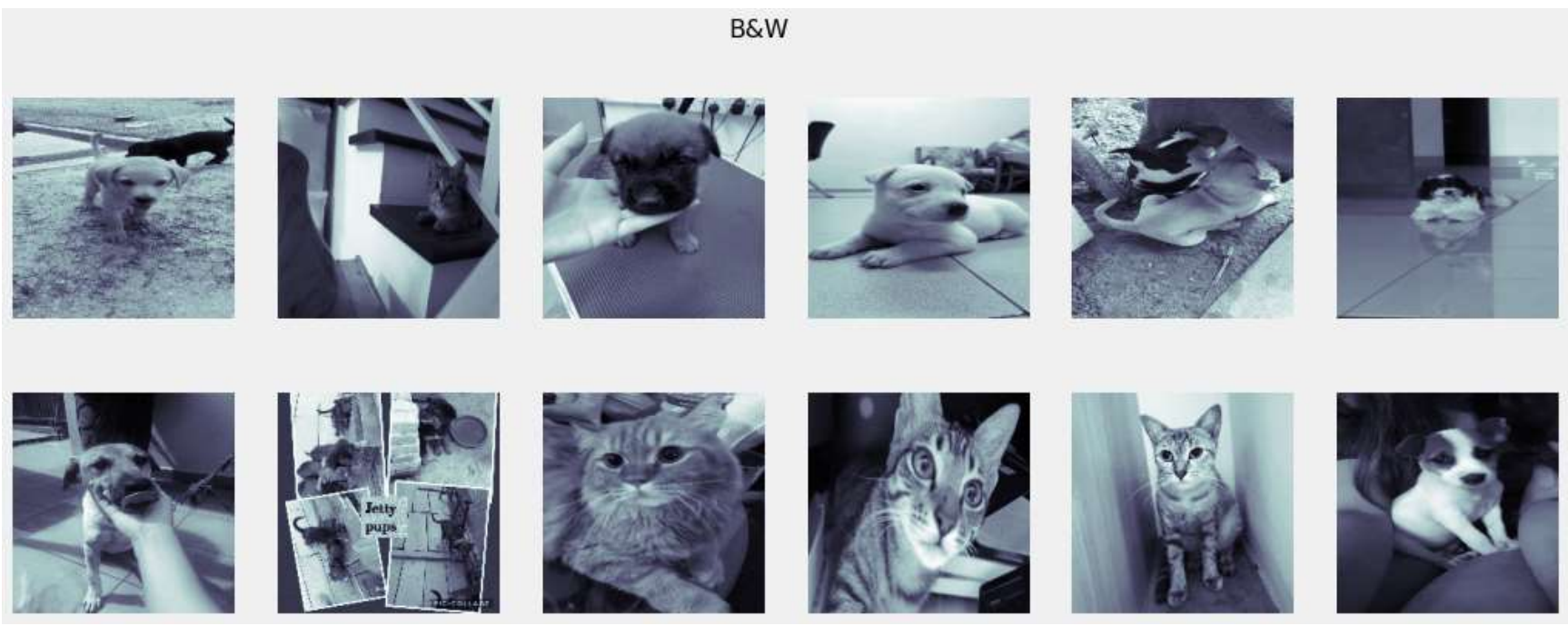
Original Images

First let us plot the original images so that the augmentations become more clear to us. After this one by one we will implement different augmentations.

In [11]:
```
plot_augments("Original", cv2.COLOR_BGR2RGB)
```


1. Black and White

In [12]:
```
plot_augments("B&W", cv2.COLOR_RGB2GRAY)
```



2. Ben Graham: Greyscale + Gaussian Blur

As in any other signals, images also can contain different types of noise, especially because of the source (camera sensor). Image Smoothing techniques help in reducing the noise.

Gaussian filters have the properties of having no overshoot to a step function input while minimizing the rise and fall time. In terms of image processing, any sharp edges in images are smoothed while minimizing too much blurring.

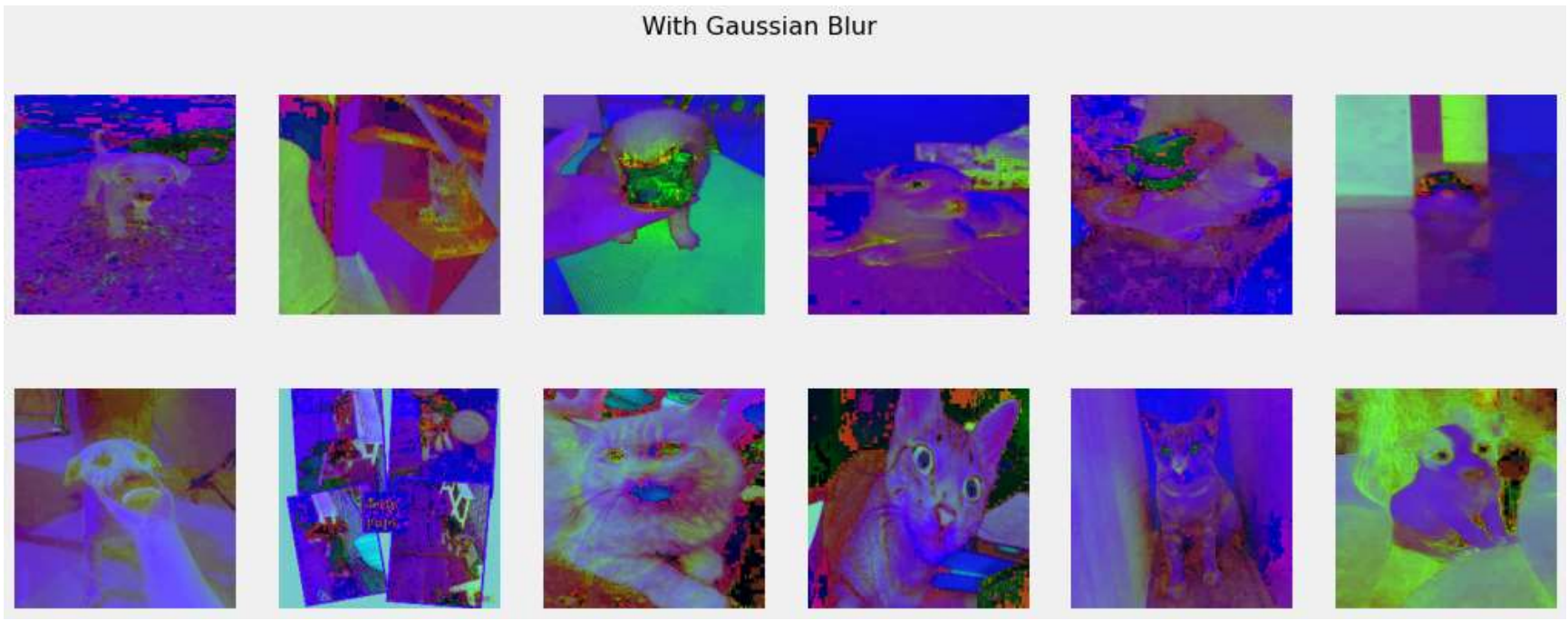
> OpenCV provides `cv2.gaussianblur()` function to apply Gaussian Smoothing on the input source image

Without Gaussian Blur

In [13]:
```
plot_augments("Without Gaussian Blur", cv2.COLOR_RGB2HSV)
```

With Gaussian Blur

In [14]:
```
plot_augments("With Gaussian Blur", cv2.COLOR_RGB2HSV)
```



---

3. Hue, Saturation, Brightness

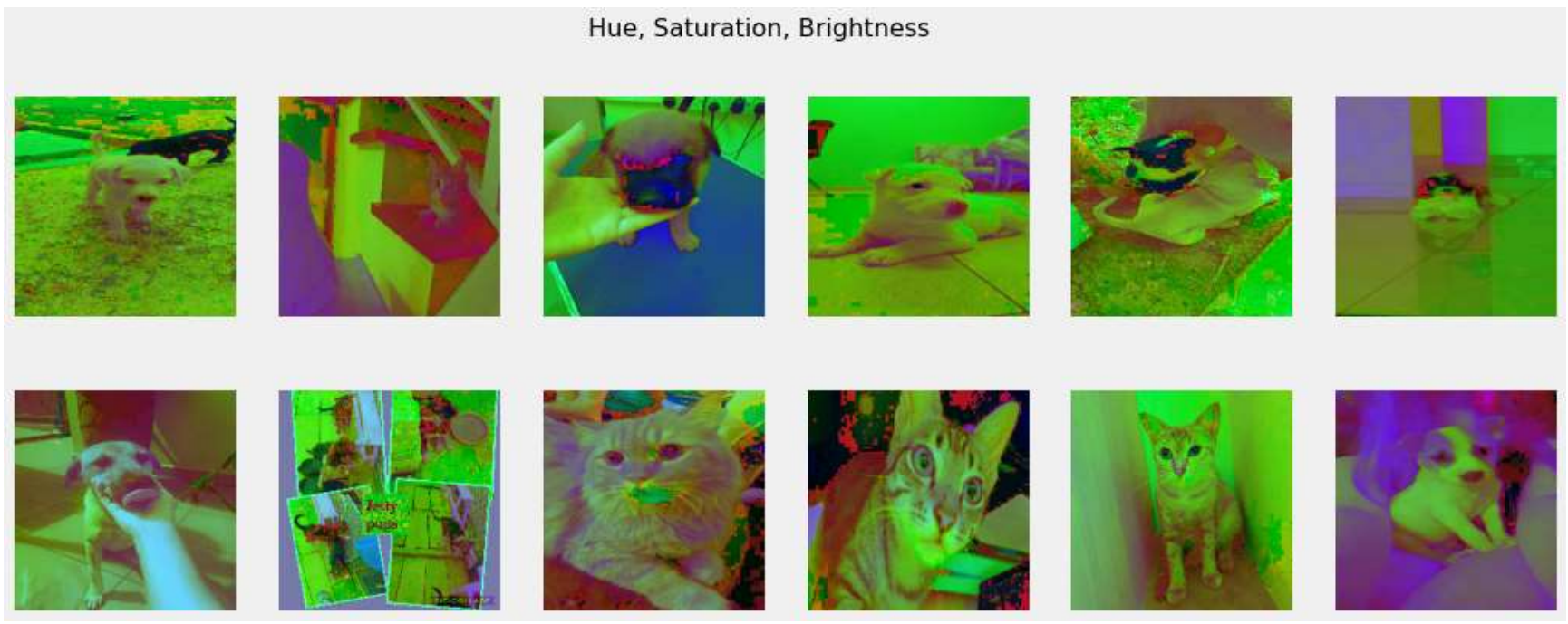
- **Hue:**
  Hue, in the context of color and graphics, refers to the attribute of a visible light due to which it is differentiated from or similar to the primary colors: red, green and blue. The term is also used to refer to colors that have no added tint or shade.

- **Saturation:**
  Color saturation refers to the intensity of color in an image. As the saturation increases, the colors appear to be more pure. As the saturation decreases, the colors appear to be more washed-out or pale.

- **Brightness:**
  Brightness is the relative lightness or darkness of a particular color, from black (no brightness) to white (full brightness). Brightness is also called Lightness in some contexts

> `cv2.COLOR_RGB2HLS` converts `RGB/BGR` to `HLS` (hue lightness saturation) with `H` range `0..180` if `8 bit` image

In [15]:
```
plot_augments("Hue, Saturation, Brightness", cv2.COLOR_RGB2HLS)
```



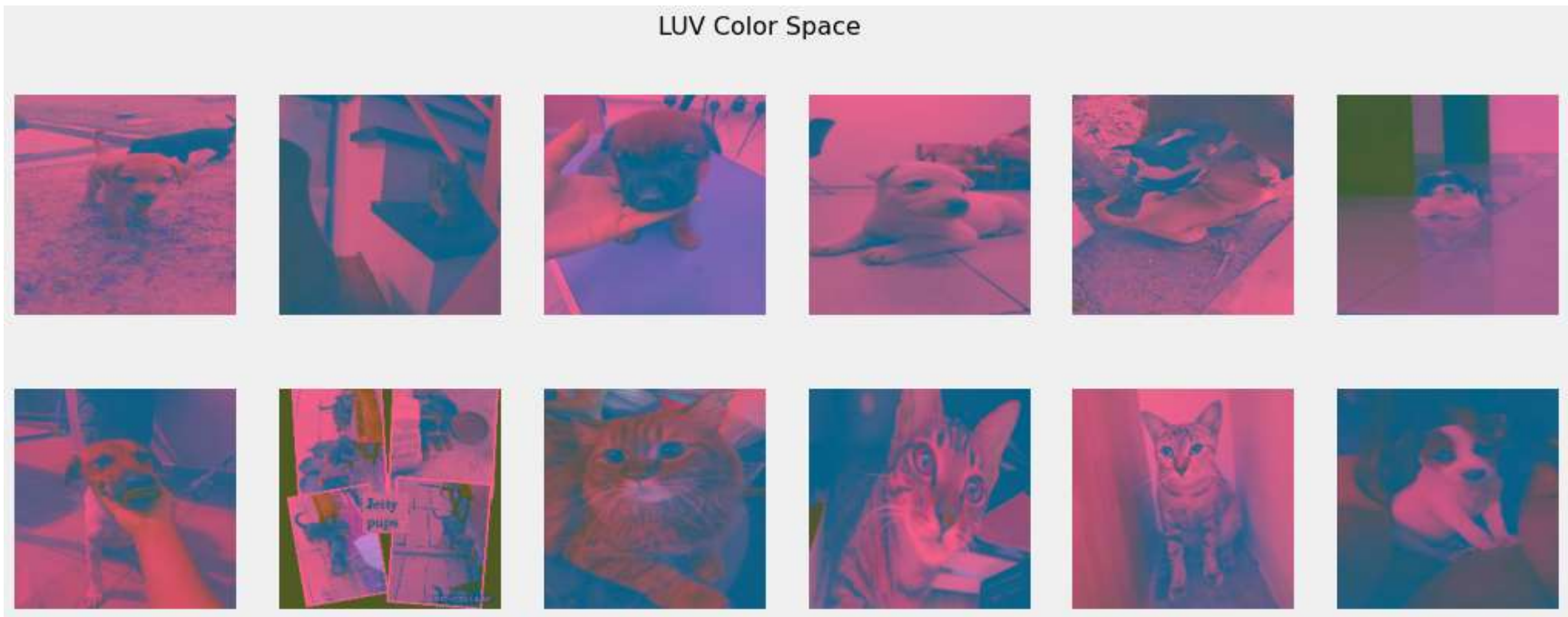
---

4. LUV Color Space

LUV decouple the "color" (chromaticity, the UV part) and "lightness" (luminance, the L part) of color.

> Hue, Saturation, Brightness `cv2.COLOR_RGB2LUV` is used to convert RGB/BGR to CIE Luv.

In [16]:
```
plot_augments("LUV Color Space", cv2.COLOR_RGB2LUV)
```



---

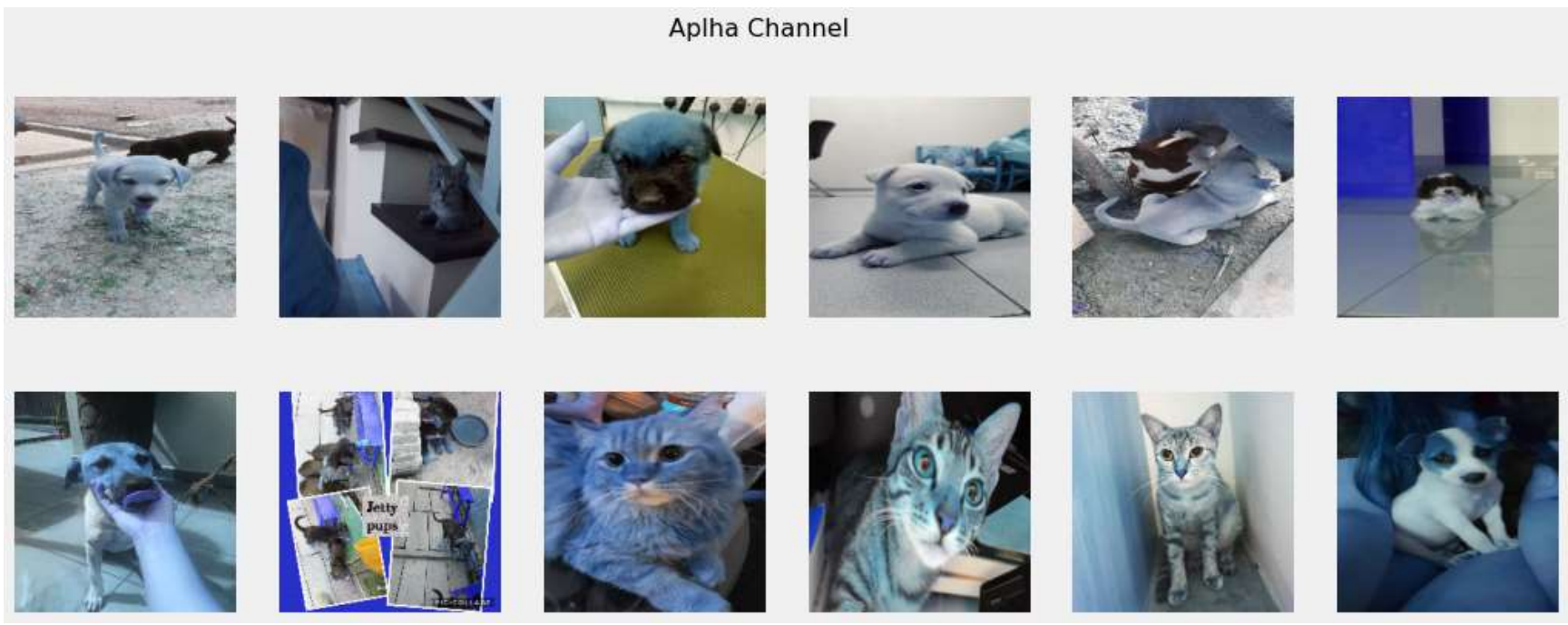
5. Alpha Channel

RGBA color values are an extension of RGB color values with an alpha channel - which specifies the opacity for a color.

An RGBA color value is specified with: `rgba(red, green, blue, alpha)`. The `alpha` parameter is a number between `0.0` (fully transparent) and `1.0` (fully opaque).

> `cv2.COLOR_RGB2RGBA` adds alpha channel to RGB or BGR image

In [17]:
```
plot_augments("Aplha Channel", cv2.COLOR_RGB2RGBA)
```



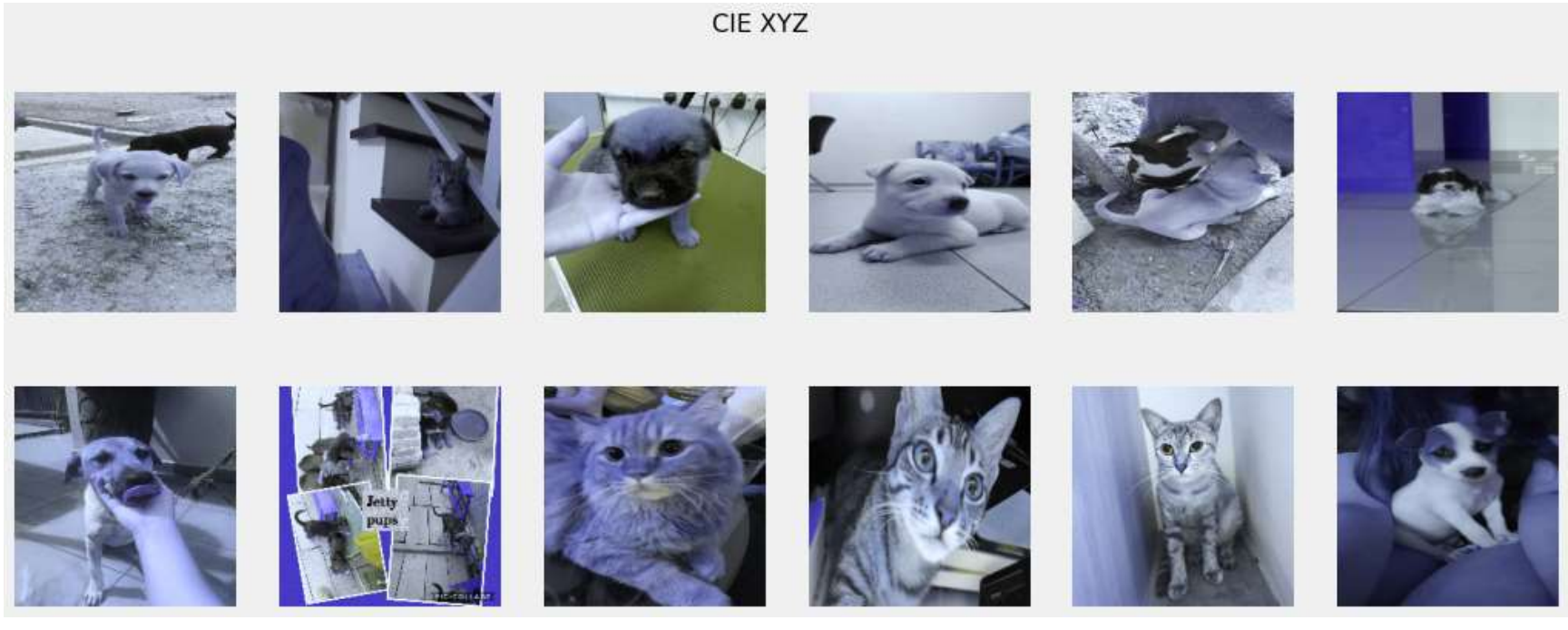
---

6. XYZ Color Space

The CIE XYZ color space encompasses all color sensations that are visible to a person with average eyesight. That is why CIE XYZ (Tristimulus values) is a device-invariant representation of color.[5] It serves as a standard reference against which many other color spaces are defined.

> `cv2.COLOR_RGB2XYZ` convert RGB/BGR to CIE XYZ

In [18]:
```
plot_augments("CIE XYZ", cv2.COLOR_RGB2XYZ)
```



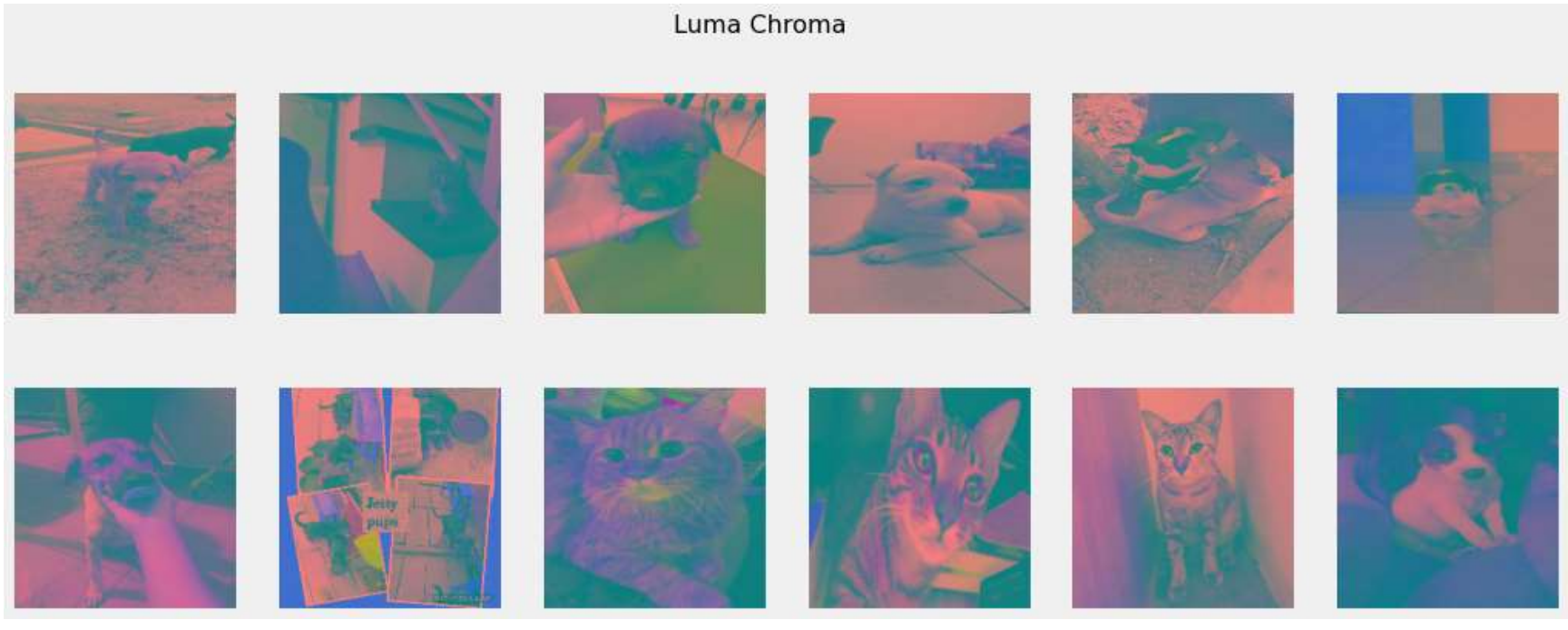
---

7. Luma Chroma

Chroma is the color and the saturation of that color. **Luma** is the brightness of the pixel.

> `cv2.COLOR_RGB2YCrCb` convert RGB/BGR to luma-chroma (aka YCC)

In [19]:
```
plot_augments("Luma Chroma", cv2.COLOR_RGB2YCrCb)
```



---

8. CIE Lab

The CIELAB color space also referred to as `L*a*b*`.

It expresses color as three values:

- `L*` for perceptual lightness, and
- `a*` and `b*` for the four unique colors of human vision: red, green, blue, and yellow.

CIELAB was intended as a perceptually uniform space, where a given numerical change corresponds to similar perceived change in color. While the LAB space is not truly perceptually uniform, it nevertheless is useful in industry for detecting small differences in color.

> `cv2.COLOR_RGB2Lab` convert RGB/BGR to CIE Lab

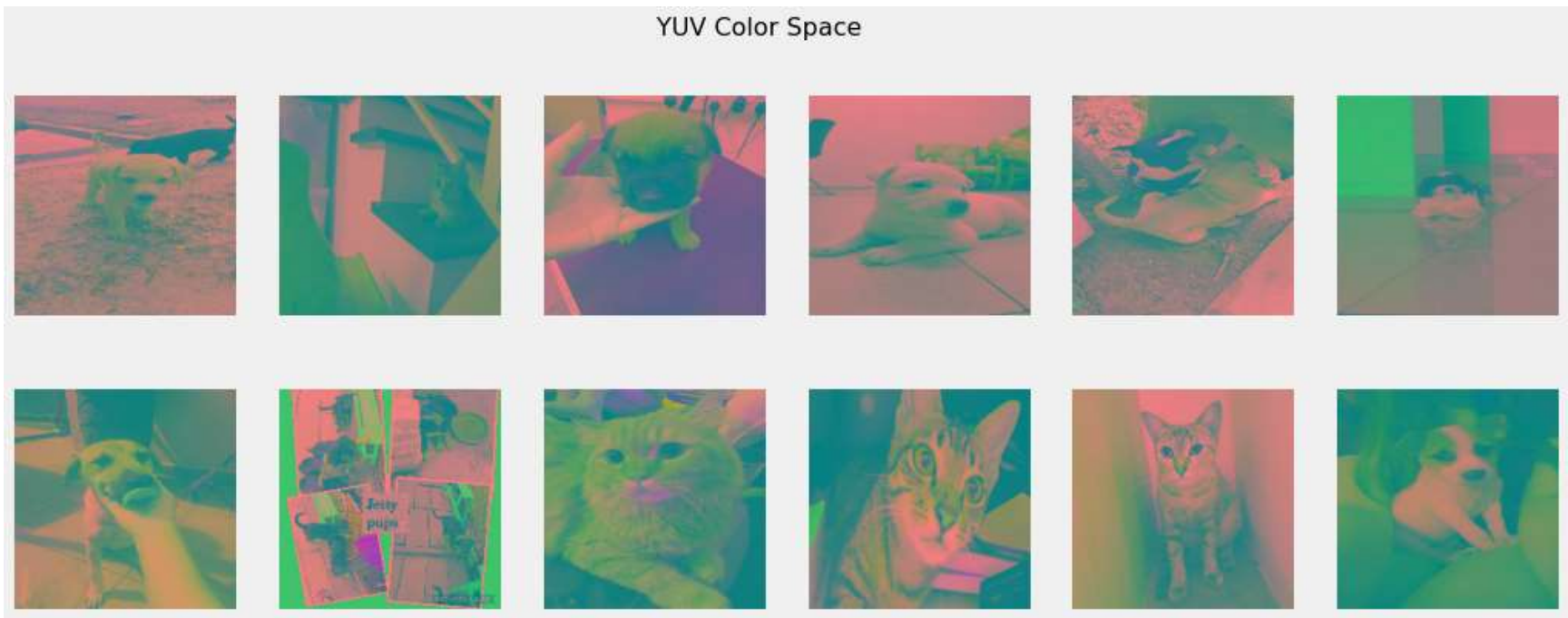
In [20]:
```
plot_augments("CIE Lab", cv2.COLOR_RGB2Lab)
```

---

9. YUV Color Space

YUV is a color encoding system typically used as part of a color image pipeline. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation.

In [21]:
```
plot_augments("YUV Color Space", cv2.COLOR_RGB2YUV)
```



Intermediate Image Augmentations

Torchvision Transforms

Transforms are common image transformations. They can be chained together using `Compose`.

Most transform classes have a function equivalent: functional transforms give fine-grained control over the transformations. This is useful if you have to build a more complex transformation pipeline (e.g. in the case of segmentation tasks).

Most transformations accept both PIL images and tensor images, although some transformations are PIL-only and some are tensor-only. The Conversion Transforms may be used to convert to and from PIL images.

The transformations that accept tensor images also accept batches of tensor images. A Tensor Image is a tensor with `(C, H, W)` shape, where `C` is a number of channels, `H` and `W` are image height and width. A batch of Tensor Images is a tensor of `(B, C, H, W)` shape, where B is a number of images in the batch.

The expected range of the values of a tensor image is implicitly defined by the tensor dtype. Tensor images with a float dtype are expected to have values in `[0, 1)`. Tensor images with an integer dtype are expected to have values in `[0, MAX_DTYPE]` where `MAX_DTYPE` is the largest value that can be represented in that dtype.

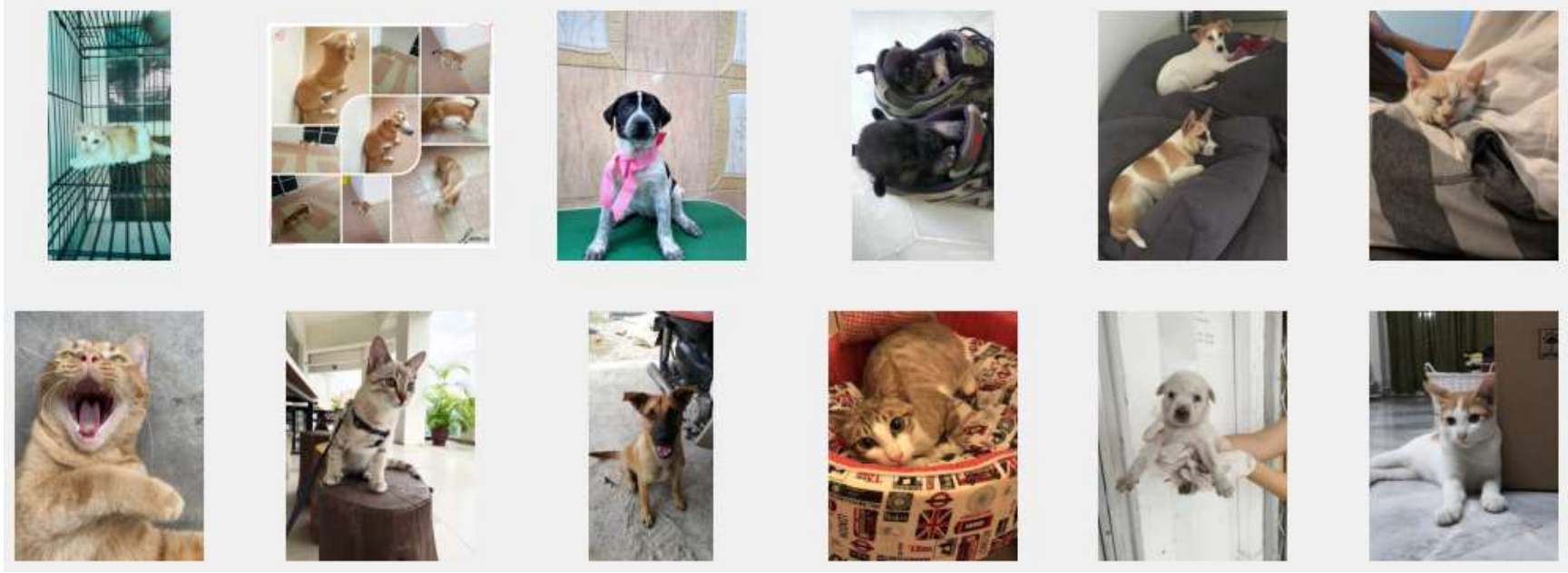
Randomized transformations will apply the same transformation to all the images of a given batch, but they will produce different transformations across calls. For reproducible transformations across calls, you may use functional transforms.

This section demonstrates 15 Torchvision Transforms with demos namely:

1. Center Crop
2. Random Crop
3. Random Resized Crop
4. Color Jitter
5. Pad

6. Random Affine
7. Random Horizontal Flip
8. Random Vertical Flip
9. Random Perspective
10. Random Rotation
11. Random Invert
12. Random Posterize
13. Random Solarize
14. Random Autocontrast
15. Random Equalize

In [22]:
```
# Select a small sample of the image paths
image_list = train.sample(12)['image_path']
image_list = image_list.reset_index()['image_path']

# Show the sample
plt.figure(figsize=(16,6))
plt.suptitle("Original View", fontsize = 16)

for k, path in enumerate(image_list):
    image = mpimg.imread(path)

    plt.subplot(2, 6, k+1)
    plt.imshow(image)
    plt.axis('off')
```



PyTorch Dataset Class

In [23]:
```
class PetDataset(Dataset):
    def __init__(self, image_list, transforms = None):
        self.image_list = image_list
        self.transforms = transforms

    def __len__(self):
        return len(self.image_list)

    def __getitem__(self, i):
        image = plt.imread(self.image_list[i])
        image = Image.fromarray(image).convert('RGB')
        image = np.asarray(image).astype(np.uint8)

        if self.transforms is not None:
            image = self.transforms(image)

        return torch.tensor(image, dtype = torch.float)
```

In [24]:
```
# Function to display applied transformations

def show_transform(image, title):
    """
    Function to Plot the Transformed Images
    """
    plt.figure(figsize = (16,6))
    plt.suptitle(title, fontsize = 16)

    #Unnormalize
    image = image / 2 + 0.5
    npimg = image.numpy()
    npimg = np.clip(npimg, 0., 1.)
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

In [25]:
```
def apply_transform(transform_layer, title):
    """
    Function to apply Torchvision Transforms to set of Images

    params: transform_layer - The transform to be applied
```

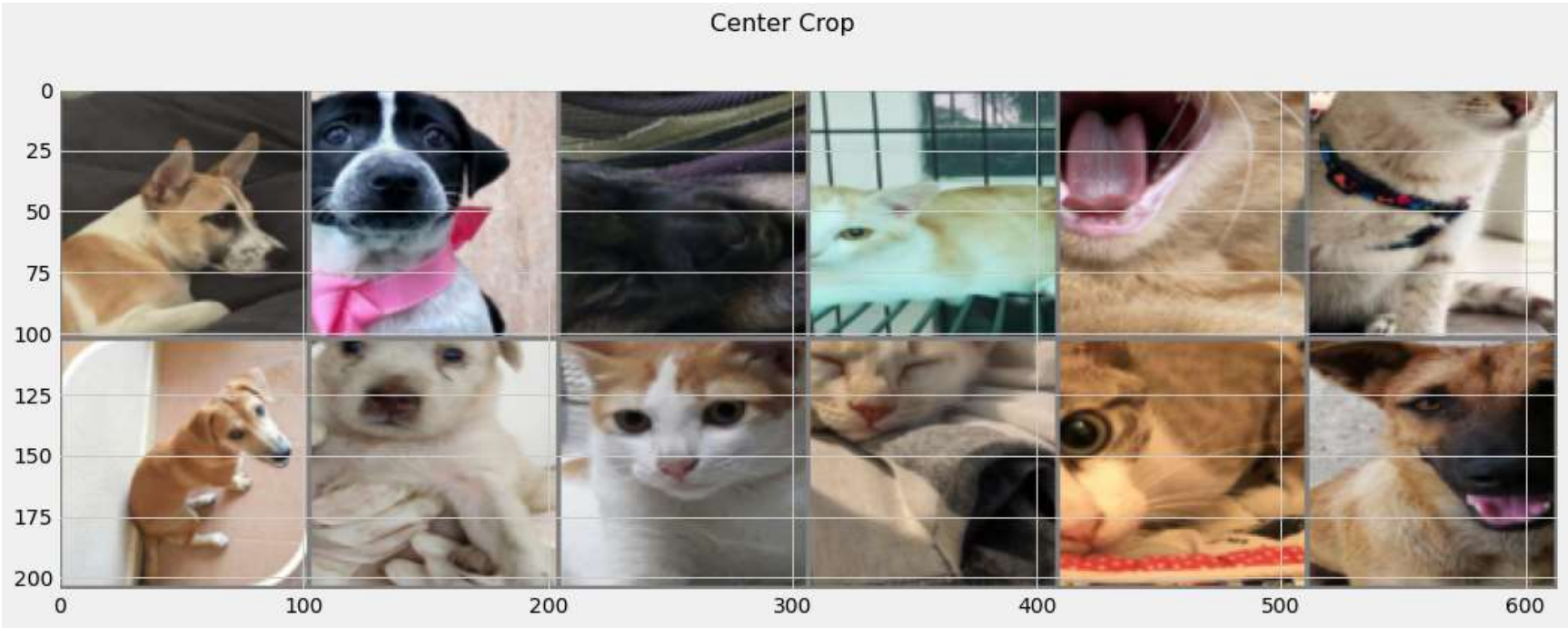
```
        title (str)     - Title of the Plot
    """
    transform = transforms.Compose(
        [
            transforms.ToPILImage(),
            transforms.Resize((300 , 300)),
            transform_layer,
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        ])

    # Dataset
    train_dataset = PetDataset(image_list = image_list, transforms = transform)

    # DataLoader
    train_dataloader = DataLoader(dataset = train_dataset, batch_size = 12, shuffle = True)

    # Select Data
    images = next(iter(train_dataloader))

    # Show Images
    show_transform(torchvision.utils.make_grid(images, nrow = 6), title = title)
```

1. Center Crop

`transforms.CenterCrop` crops the given image at the center.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions.
- If image size is smaller than output size along any edge, image is padded with 0 and then center cropped.

In [26]:
```
apply_transform(transforms.CenterCrop((100, 100)), "Center Crop")
```



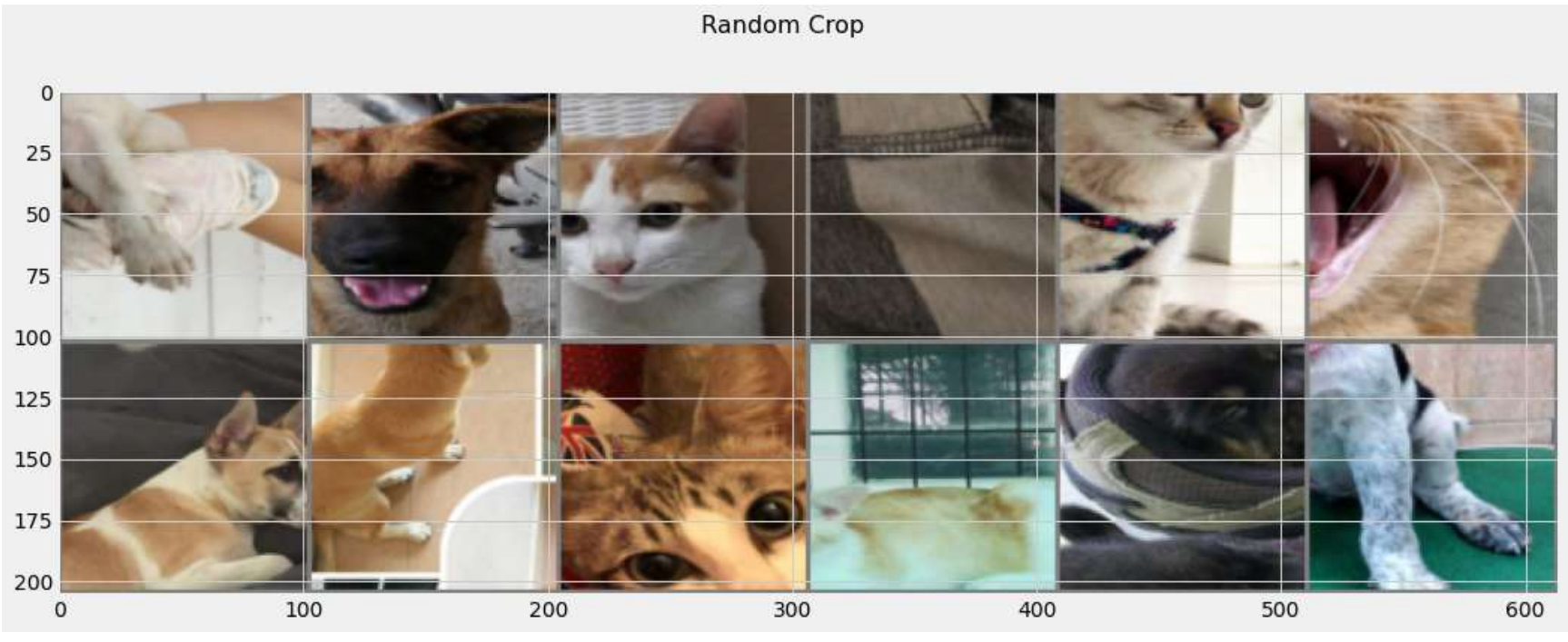
---

2. Random Crop

`transforms.RandomCrop` Crop the given image at a random location.

- If the image is torch Tensor, it is expected to have [..., H, W] shape, where ... means an arbitrary number of leading dimensions, but if non-constant padding is used, the input is expected to have at most 2 leading dimensions

In [27]:
```
apply_transform(transforms.RandomCrop((100, 100)), "Random Crop")
```



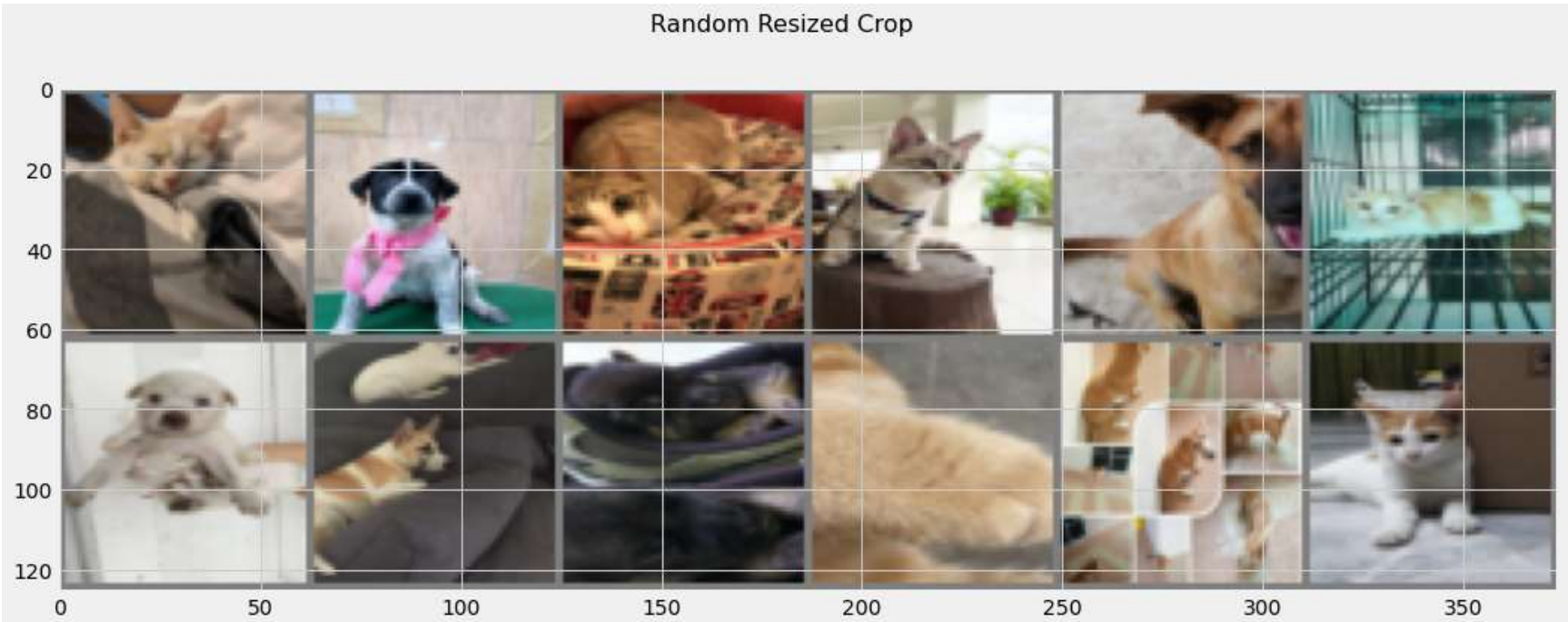
3. Random Resized Crop

`transforms.RandomResizedCrop` Crops a random portion of image and resize it to a given size.

- If the image is torch Tensor, it is expected to have [..., H, W] shape, where ... means an arbitrary number of leading dimensions

A crop of the original image is made: the crop has a random area `(H * W)` and a random aspect ratio. This crop is finally resized to the given size. This is popularly used to train the **Inception networks.**

In [28]:
```
apply_transform(transforms.RandomResizedCrop(size = 60), "Random Resized Crop")
```



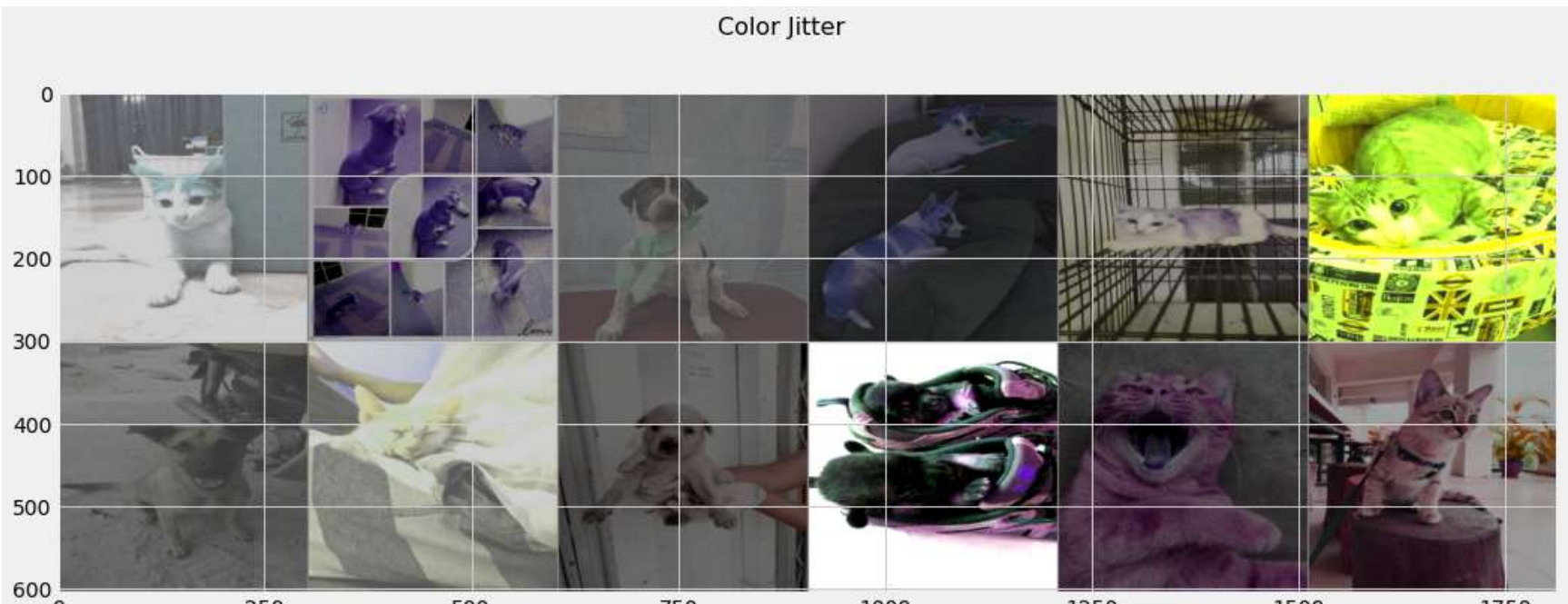
---

4. Color Jitter

`transforms.ColorJitter()` randomly change the brightness, contrast, saturation and hue of an image.

- If the image is torch Tensor, it is expected to have `[..., 1 or 3, H, W]` shape, where `...` means an arbitrary number of leading dimensions.
- If img is PIL Image, mode `“1”, “I”, “F”` and modes with transparency (alpha channel) are not supported.

In [29]:
```
apply_transform(transforms.ColorJitter(brightness=0.7, contrast=0.7, saturation=0.7, hue=0.5), "Color Jitter")
```



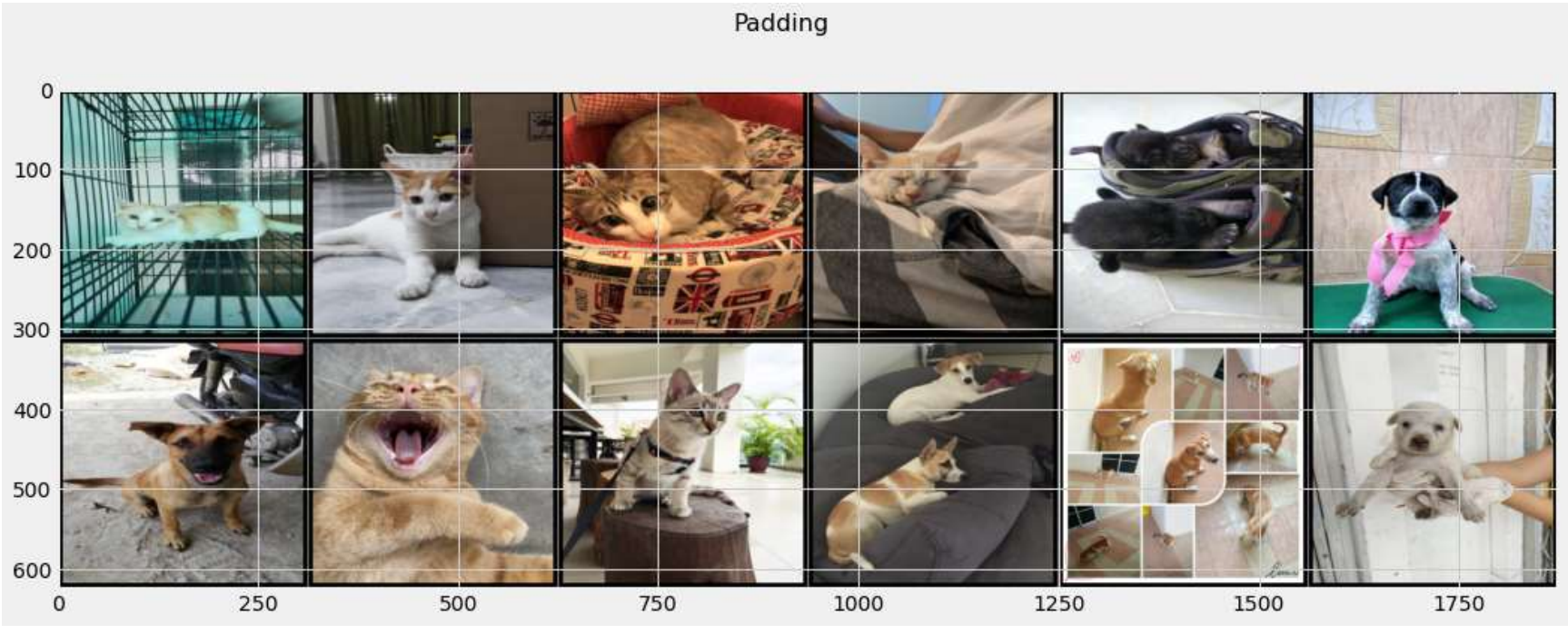
---

5. Pad

`transforms.Pad()` pad the given image on all sides with the given “pad” value.

- If the image is torch Tensor, it is expected to have `[..., H, W]` shape, where `...` means at most 2 leading dimensions for mode reflect and symmetric, at most 3 leading dimensions for mode edge, and an arbitrary number of leading dimensions for mode constant

In [30]:
```
apply_transform(transforms.Pad(padding = 5), "Padding")
```



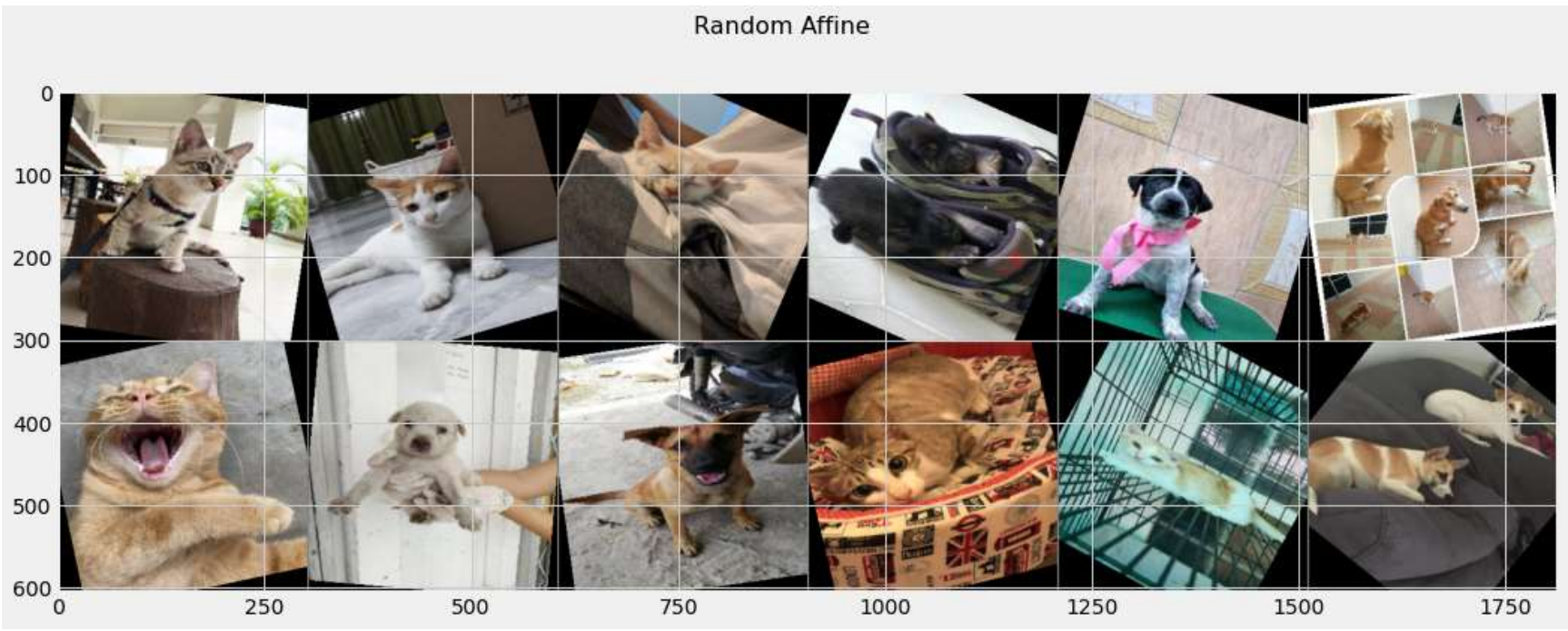
---

6. Random Affine

`transforms.RandomAffine()` applies Random affine transformation of the image keeping center invariant.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions.

In [31]:
```
apply_transform(transforms.RandomAffine(degrees=45), "Random Affine")
```



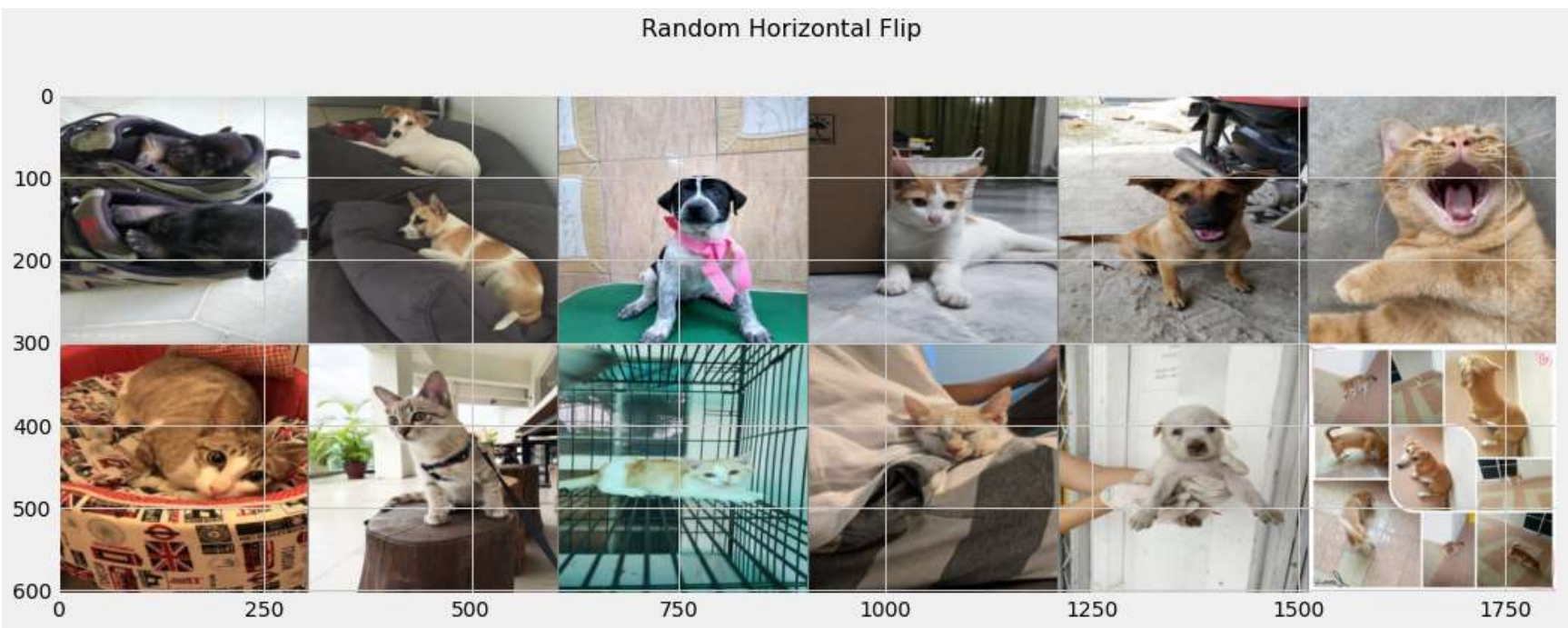
---

7. Random Horizontal Flip

`transforms.RandomHorizontalFlip` Horizontally flips the given image randomly with a given probability.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions

In [32]:
```
apply_transform(transforms.RandomHorizontalFlip(p=0.7), "Random Horizontal Flip")
```



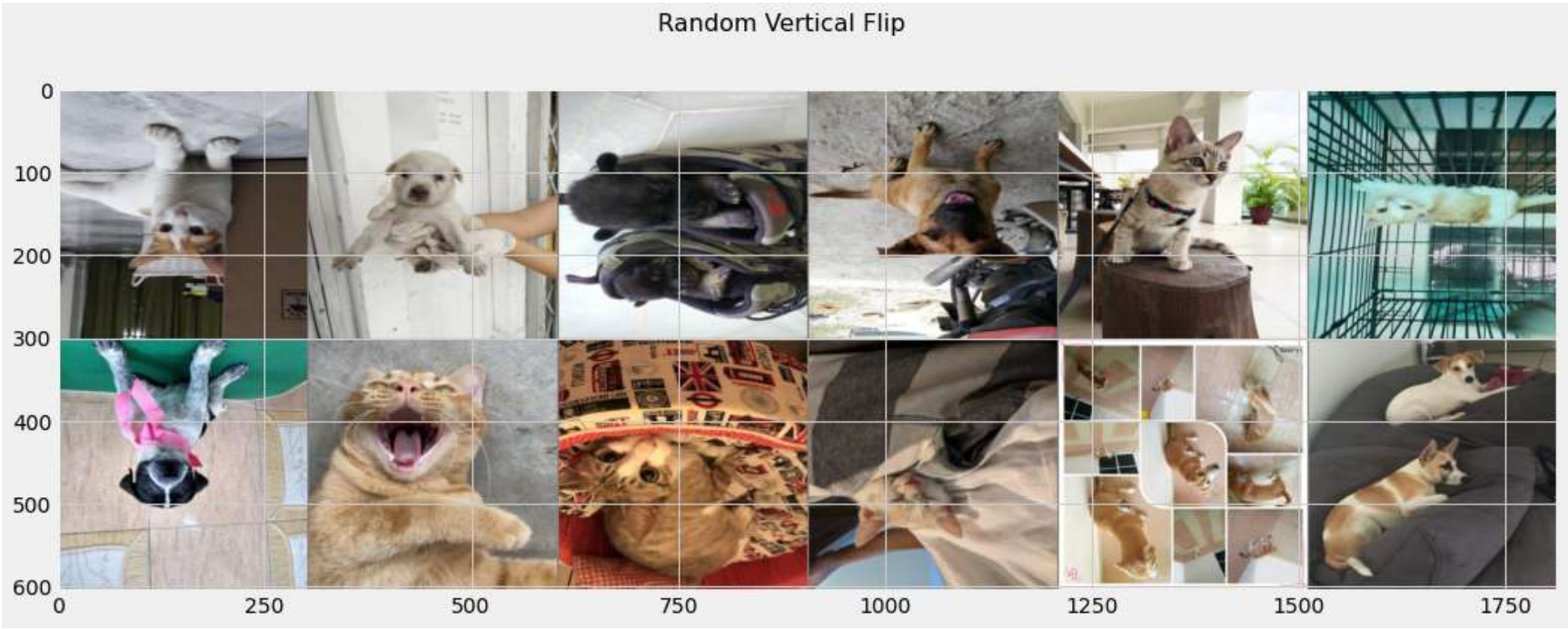
---

8. Random Vertical Flip

`transforms.RandomVerticalFlip()` Vertically flip the given image randomly with a given probability.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions.

In [33]:
```
apply_transform(transforms.RandomVerticalFlip(p=0.7), "Random Vertical Flip")
```

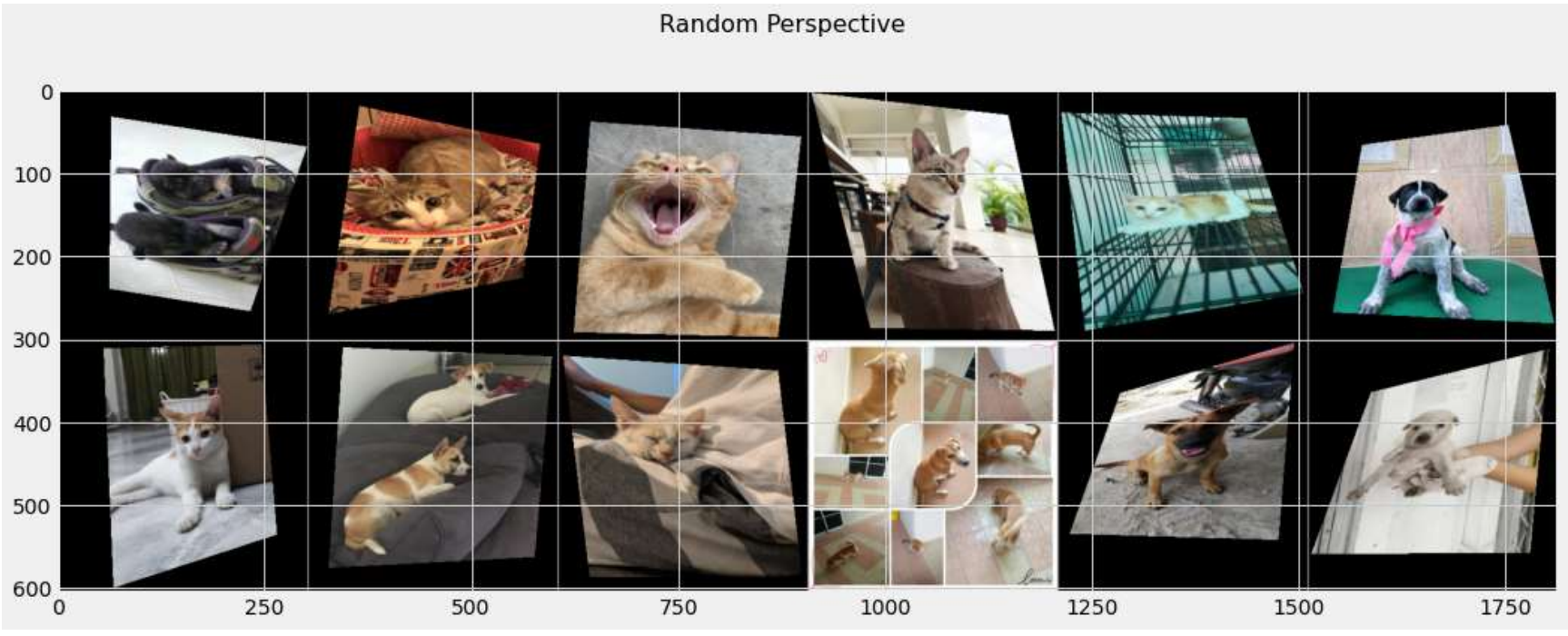


9. Random Perspective

`transforms.RandomPerspective()` Performs a random perspective transformation of the given image with a given probability.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions.

In [34]:
```
apply_transform(transforms.RandomPerspective(p=0.7), "Random Perspective")
```

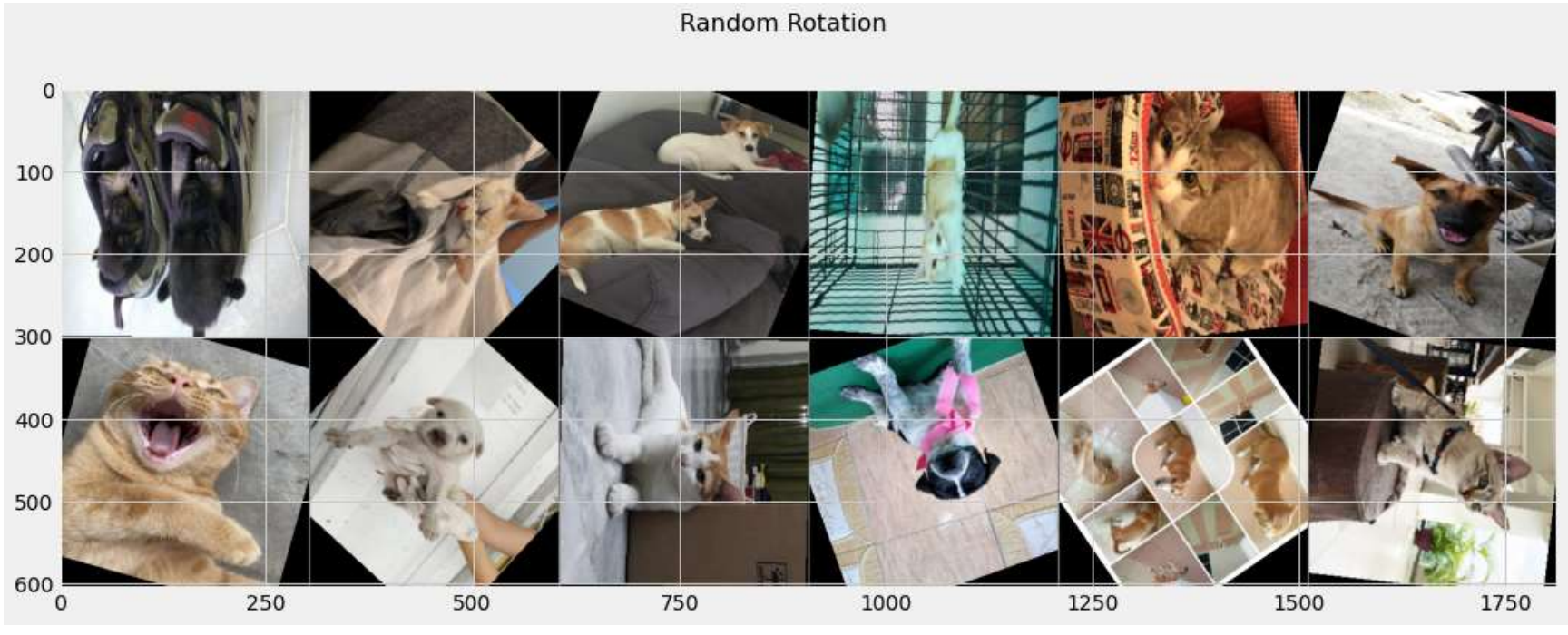


10. Random Rotation

`transforms.RandomRotation()` Rotates the image by angle.

- If the image is torch Tensor, it is expected to have `[…, H, W]` shape, where `…` means an arbitrary number of leading dimensions.

In [35]:
```
apply_transform(transforms.RandomRotation(degrees = 180), "Random Rotation")
```



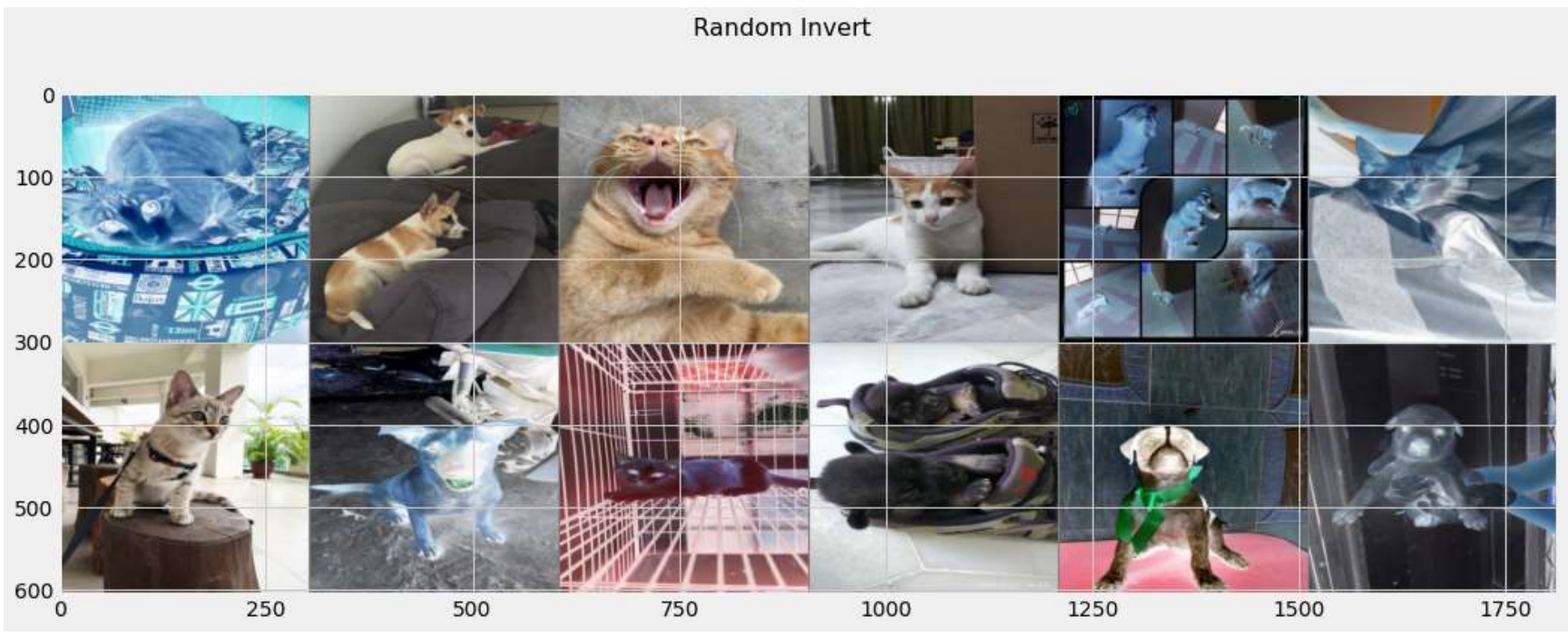
---

11. Random Invert

`transforms.RandomInvert()` Inverts the colors of the given image randomly with a given probability.

- If img is a Tensor, it is expected to be in `[…, 1 or 3, H, W]` format, where `…` means it can have an arbitrary number of leading dimensions.
- If img is PIL Image, it is expected to be in mode “L” or “RGB”.

In [36]:
```
apply_transform(transforms.RandomInvert(p=0.7), "Random Invert")
```



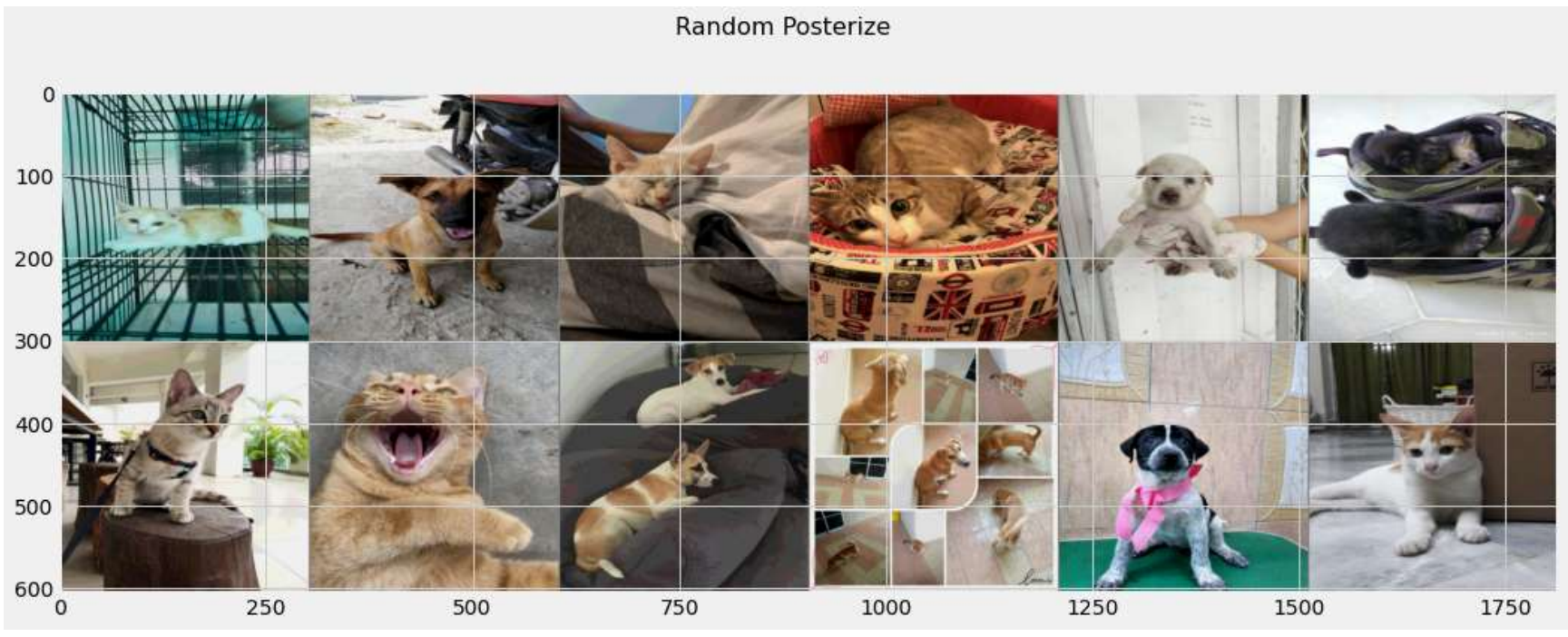
---

12. Random Posterize

`transforms.RandomPosterize()` Posterize the image randomly with a given probability by reducing the number of bits for each color channel.

- If the image is torch Tensor, it should be of type `torch.uint8`, and it is expected to have `[…, 1 or 3, H, W]` shape, where `…` means an arbitrary number of leading dimensions. - If img is PIL Image, it is expected to be in mode “L” or “RGB”.

In [37]:
```
apply_transform(transforms.RandomPosterize(bits = 4, p=0.7), "Random Posterize")
```

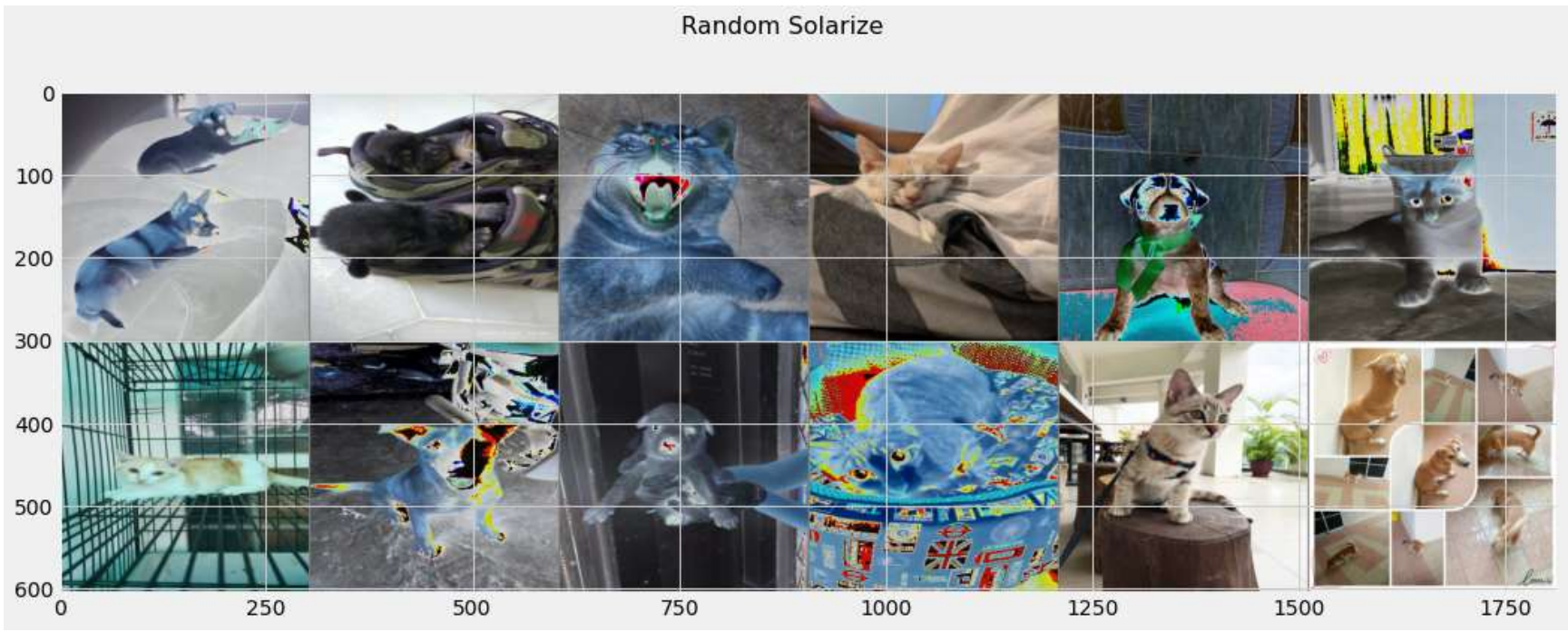


13. Random Solarize

`transforms.RandomSolarize()` Solarizes the image randomly with a given probability by inverting all pixel values above a threshold.

- If img is a Tensor, it is expected to be in `[…, 1 or 3, H, W]` format, where `…` means it can have an arbitrary number of leading dimensions.
- If img is PIL Image, it is expected to be in mode “L” or “RGB”.

In [38]:
```
apply_transform(transforms.RandomSolarize(threshold = 30, p=0.7), "Random Solarize")
```

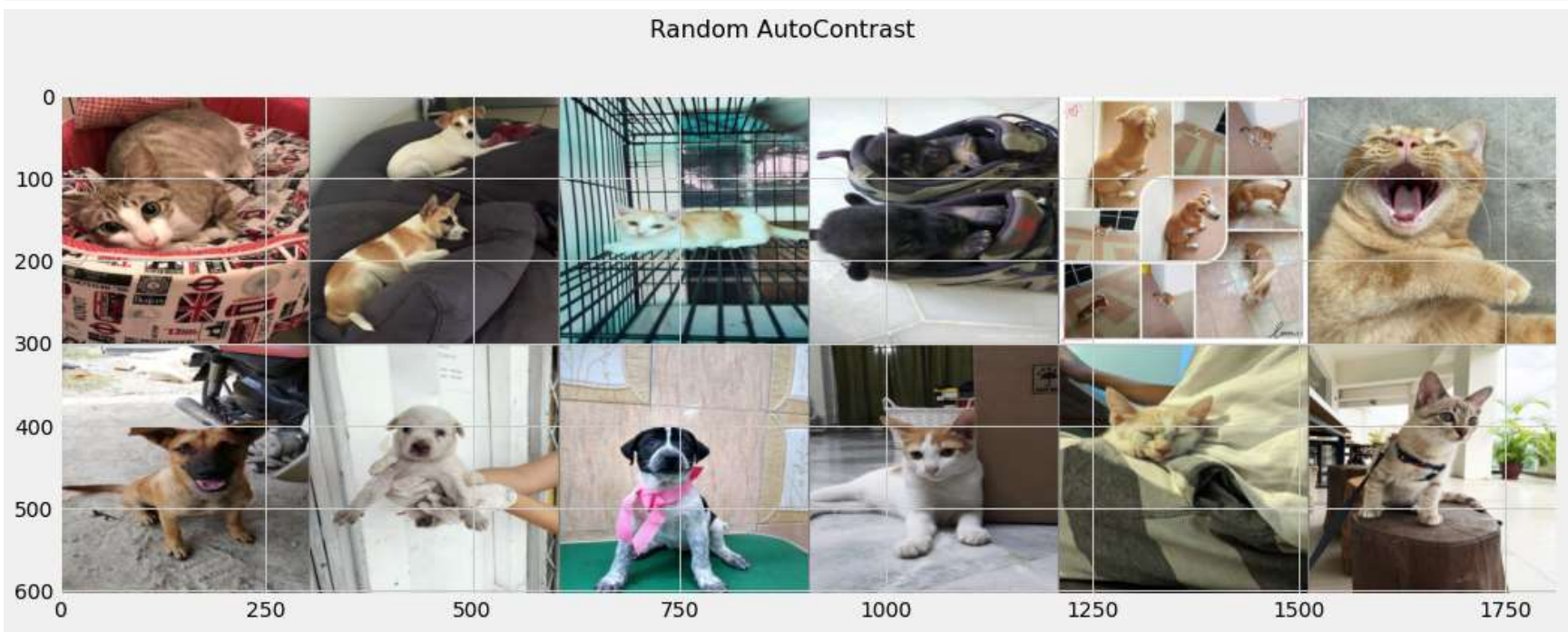


14. Random Auto Contrast

`transforms.RandomAutocontrast()` Autocontrasts the pixels of the given image randomly with a given probability.

- If the image is torch Tensor, it is expected to have `[…, 1 or 3, H, W]` shape, where `…` means an arbitrary number of leading dimensions.
- If img is PIL Image, it is expected to be in mode “L” or “RGB”.

In [39]:
```
apply_transform(transforms.RandomAutocontrast(p=0.7), "Random AutoContrast")
```

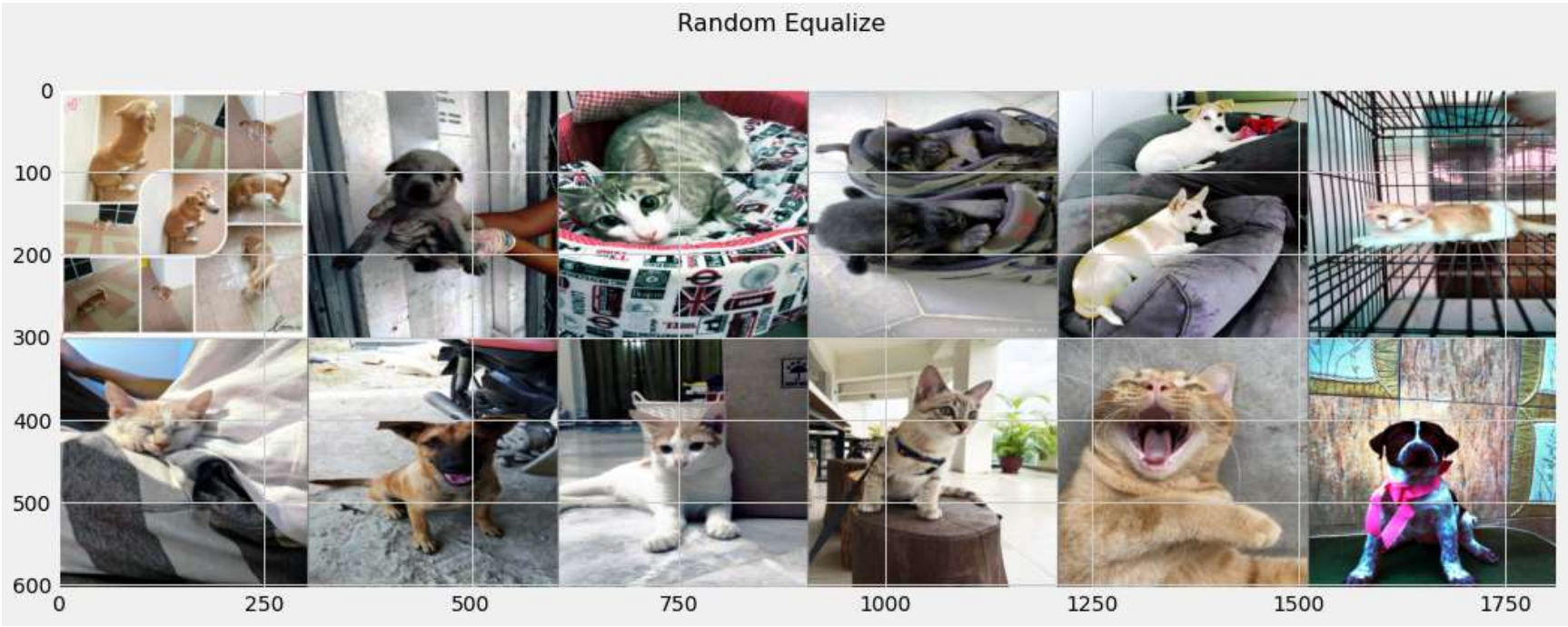


15. Random Equalize

`transforms.RandomEqualize()` Equalizes the histogram of the given image randomly with a given probability.

- If the image is torch Tensor, it is expected to have `[…, 1 or 3, H, W]` shape, where `…` means an arbitrary number of leading dimensions.
- If img is PIL Image, it is expected to be in mode “P”, “L” or “RGB”.

In [40]:
```
apply_transform(transforms.RandomEqualize(p=0.7), "Random Equalize")
```



Advanced Data Augmentations

The idea is to generate new and realistic fetures based on labels. GANs are excellent at generating realistic data. We can condition this generation by using Conditional Generative Adversarial Networks

In [41]:
```
latent_dim = 100 # dimension of the latent space
n_samples = 1000 # size of our dataset
n_classes = 3
n_features = 2 # we use 2 features since we'd like to visualize them
```

We start by creating random clusters of points, n_classes, with features, n_features. We make use of make_blobs from scikit learn that generates gaussian blobs

In [42]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=n_features, random_state=123)

print('Size of our dataset:', len(X))
print('Number of features:', X.shape[1])
print('Classes:', set(y))
```

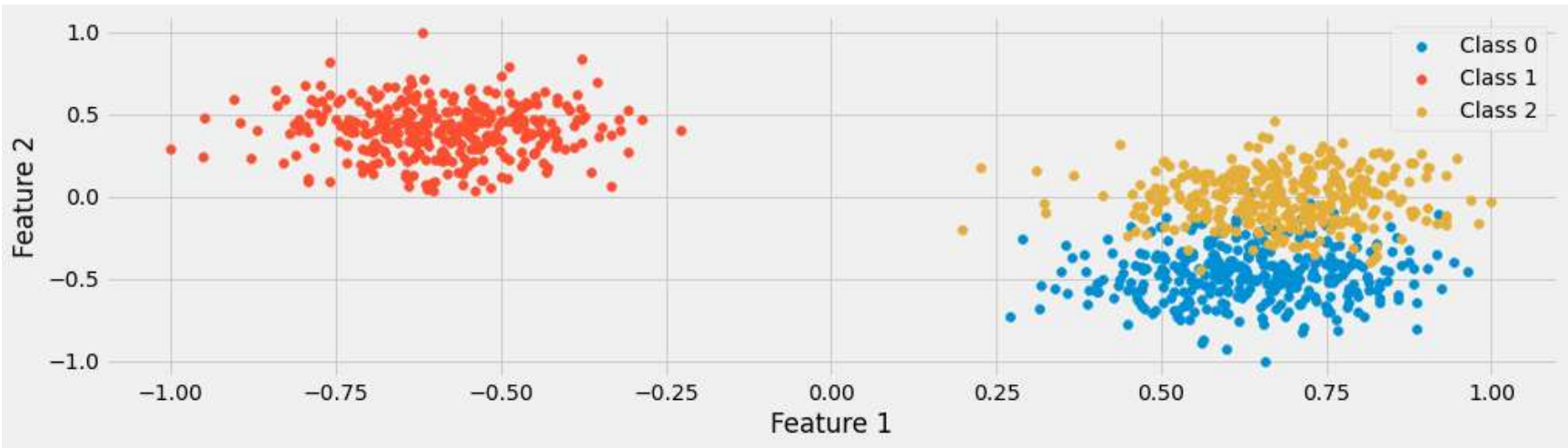
```
Size of our dataset: 1000
Number of features: 2
Classes: {0, 1, 2}
```

Following we normalize our features to help with the learning

In [43]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-1, 1))

scaled_X = scaler.fit_transform(X)
```

In [44]:
```
fig, ax = plt.subplots(figsize=(15, 4))
legend = []

for i in range(n_classes):
    plt.scatter(scaled_X[:, 0][np.where(y==i)], scaled_X[:, 1][np.where(y==i)], )
    legend.append('Class %d' % i)

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.legend(legend)
```

Out[44]:
```
<matplotlib.legend.Legend at 0x7be8ceab7dd0>
```



In [45]:
```
def build_discriminator(optimizer=Adam(0.0002, 0.5)):
    '''
    Defines and compiles discriminator model.
    This architecture has been inspired by:
    https://github.com/eriklindernoren/Keras-GAN/blob/master/cgan/cgan.py
    and adapted for this problem.
```

```
    Params:
        optimizer=Adam(0.0002, 0.5) - recommended values
    '''
    features = Input(shape=(n_features,))
    label = Input(shape=(1,), dtype='int32')

    # Using an Embedding layer is recommended by the papers
    label_embedding = Flatten()(Embedding(n_classes, n_features)(label))

    # We condition the discrimination of generated features
    inputs = multiply([features, label_embedding])

    x = Dense(512)(inputs)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dense(512)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.4)(x)
    x = Dense(512)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.4)(x)

    valid = Dense(1, activation='sigmoid')(x)

    model = Model([features, label], valid)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    model.summary()

    return model
```

In [46]:

```
def build_generator():
    '''
    Defines the generator model.
    This architecture has been inspired by:
    https://github.com/eriklindernoren/Keras-GAN/blob/master/cgan/cgan.py
    and adapted for this problem.
    '''

    noise = Input(shape=(latent_dim,))
    label = Input(shape=(1,), dtype='int32')

    # Using an Embedding layer is recommended by the papers
    label_embedding = Flatten()(Embedding(n_classes, latent_dim)(label))

    # We condition the generation of features
    inputs = multiply([noise, label_embedding])

    x = Dense(256)(inputs)
    x = LeakyReLU(alpha=0.2)(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = Dense(512)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = Dense(1024)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = BatchNormalization(momentum=0.8)(x)

    features = Dense(n_features, activation='tanh')(x)

    model = Model([noise, label], features)
    model.summary()

    return model
```

In [47]:

```
def build_gan(generator, discriminator, optimizer=Adam(0.0002, 0.5)):
    '''
    Defines and compiles GAN model. It bassically chains Generator
    and Discriminator in an assembly-line sort of way where the input is
    the Generator's input. The Generator's output is the input of the Discriminator,
    which outputs the output of the whole GAN.

    Params:
        optimizer=Adam(0.0002, 0.5) - recommended values
    '''

    noise = Input(shape=(latent_dim,))
    label = Input(shape=(1,))

    features = generator([noise, label])
    valid = discriminator([features, label])

    # We freeze the discriminator's layers since we're only
    # interested in the generator and its learning
    discriminator.trainable = False

    model = Model([noise, label], valid)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    model.summary()

    return model
```

In [48]:

```
discriminator = build_discriminator()
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 1)]	0	
embedding (Embedding)	(None, 1, 2)	6	input_2[0][0]
input_1 (InputLayer)	[(None, 2)]	0	
flatten (Flatten)	(None, 2)	0	embedding[0][0]
multiply (Multiply)	(None, 2)	0	input_1[0][0] flatten[0][0]
dense (Dense)	(None, 512)	1536	multiply[0][0]
leaky_re_lu (LeakyReLU)	(None, 512)	0	dense[0][0]
dense_1 (Dense)	(None, 512)	262656	leaky_re_lu[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0	dense_1[0][0]
dropout (Dropout)	(None, 512)	0	leaky_re_lu_1[0][0]
dense_2 (Dense)	(None, 512)	262656	dropout[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 512)	0	dense_2[0][0]
dropout_1 (Dropout)	(None, 512)	0	leaky_re_lu_2[0][0]
dense_3 (Dense)	(None, 1)	513	dropout_1[0][0]

```
Total params: 527,367
Trainable params: 527,367
Non-trainable params: 0
```

In [49]:
```
generator = build_generator()
```

```
Model: "model_1"
```

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	[(None, 1)]	0	
embedding_1 (Embedding)	(None, 1, 100)	300	input_4[0][0]
input_3 (InputLayer)	[(None, 100)]	0	
flatten_1 (Flatten)	(None, 100)	0	embedding_1[0][0]
multiply_1 (Multiply)	(None, 100)	0	input_3[0][0] flatten_1[0][0]
dense_4 (Dense)	(None, 256)	25856	multiply_1[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 256)	0	dense_4[0][0]
batch_normalization (BatchNorma	(None, 256)	1024	leaky_re_lu_3[0][0]
dense_5 (Dense)	(None, 512)	131584	batch_normalization[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 512)	0	dense_5[0][0]
batch_normalization_1 (BatchNor	(None, 512)	2048	leaky_re_lu_4[0][0]
dense_6 (Dense)	(None, 1024)	525312	batch_normalization_1[0][0]
leaky_re_lu_5 (LeakyReLU)	(None, 1024)	0	dense_6[0][0]
batch_normalization_2 (BatchNor	(None, 1024)	4096	leaky_re_lu_5[0][0]
dense_7 (Dense)	(None, 2)	2050	batch_normalization_2[0][0]

```
Total params: 692,270
Trainable params: 688,686
Non-trainable params: 3,584
```

In [50]:
```
gan = build_gan(generator, discriminator)
```

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_5 (InputLayer)            [(None, 100)]        0
__________________________________________________________________________________________________
input_6 (InputLayer)            [(None, 1)]          0
__________________________________________________________________________________________________
model_1 (Functional)            (None, 2)            692270      input_5[0][0]
                                                                 input_6[0][0]
__________________________________________________________________________________________________
model (Functional)              (None, 1)            527367      model_1[0][0]
                                                                 input_6[0][0]
==================================================================================================
Total params: 1,219,637
Trainable params: 688,686
Non-trainable params: 530,951
__________________________________________________________________________________________________
```

In [51]:

```python
def get_random_batch(X, y, batch_size):
    '''
    Will return random batches of size batch_size

    Params:
        X: numpy array - features
        y: numpy array - classes
        batch_size: Int
    '''
    idx = np.random.randint(0, len(X))

    X_batch = X[idx:idx+batch_size]
    y_batch = y[idx:idx+batch_size]

    return X_batch, y_batch
```

In [52]:

```python
def train_gan(gan, generator, discriminator,
              X, y,
              n_epochs=100, batch_size=32,
              hist_every=10, log_every=1):
    '''
    Trains discriminator and generator (last one through the GAN)
    separately in batches of size batch_size. The training goes as follow:
        1. Discriminator is trained with real features from our training data
        2. Discriminator is trained with fake features generated by the Generator
        3. GAN is trained, which will only change the Generator's weights.

    Params:
        gan: GAN model
        generator: Generator model
        discriminator: Discriminator model
        X: numpy array - features
        y: numpy array - classes
        n_epochs: Int
        batch_size: Int
        hist_every: Int - will save the training loss and accuracy every hist_every epochs
        log_every: Int - will output the loss and accuracy every log_every epochs

    Returns:
        loss_real_hist: List of Floats
        acc_real_hist: List of Floats
        loss_fake_hist: List of Floats
        acc_fake_hist: List of Floats
        loss_gan_hist: List of Floats
        acc_gan_hist: List of Floats
    '''

    half_batch = int(batch_size / 2)

    acc_real_hist = []
    acc_fake_hist = []
    acc_gan_hist = []
    loss_real_hist = []
    loss_fake_hist = []
    loss_gan_hist = []

    # Initialize W&B
    run = wandb.init(project='PetFinder-GANs',
                config= WANDB_CONFIG)

    for epoch in range(n_epochs):

        X_batch, labels = get_random_batch(X, y, batch_size)

        # train with real values
        y_real = np.ones((X_batch.shape[0], 1))
        loss_real, acc_real = discriminator.train_on_batch([X_batch, labels], y_real)

        # train with fake values
        noise = np.random.uniform(0, 1, (labels.shape[0], latent_dim))
        X_fake = generator.predict([noise, labels])
        y_fake = np.zeros((X_fake.shape[0], 1))
        loss_fake, acc_fake = discriminator.train_on_batch([X_fake, labels], y_fake)

        y_gan = np.ones((labels.shape[0], 1))
        loss_gan, acc_gan = gan.train_on_batch([noise, labels], y_gan)
```

```
        if (epoch+1) % hist_every == 0:
            acc_real_hist.append(acc_real)
            acc_fake_hist.append(acc_fake)
            acc_gan_hist.append(acc_gan)
            loss_real_hist.append(loss_real)
            loss_fake_hist.append(loss_fake)
            loss_gan_hist.append(loss_gan)

            wandb.log({'acc_real': acc_real,
                       'acc_fake': acc_fake,
                       'acc_gan': acc_gan,
                       'loss_real': loss_real,
                       'loss_fake': loss_fake,
                       'loss_gan': loss_gan,
                      })

        if (epoch+1) % log_every == 0:
            lr = 'loss real: {:.3f}'.format(loss_real)
            ar = 'acc real: {:.3f}'.format(acc_real)
            lf = 'loss fake: {:.3f}'.format(loss_fake)
            af = 'acc fake: {:.3f}'.format(acc_fake)
            lg = 'loss gan: {:.3f}'.format(loss_gan)
            ag = 'acc gan: {:.3f}'.format(acc_gan)

            print('{}, {} | {}, {} | {}, {}'.format(lr, ar, lf, af, lg, ag))

    # Close W&B run
    wandb.finish()

    return loss_real_hist, acc_real_hist, loss_fake_hist, acc_fake_hist, loss_gan_hist, acc_gan_hist
```

In [53]:
```
loss_real_hist, acc_real_hist, loss_fake_hist, acc_fake_hist, loss_gan_hist, acc_gan_hist = train_gan(gan, generator, d
```

```
wandb: Currently logged in as: leozhao. Use `wandb login --relogin` to force relogin
```

Tracking run with wandb version 0.15.12

Run data is saved locally in /kaggle/working/wandb/run-20231030_213336-czli7xjp

Syncing run **ritualistic-possession-1** to Weights & Biases (docs)

View project at https://wandb.ai/leozhao/PetFinder-GANs

View run at https://wandb.ai/leozhao/PetFinder-GANs/runs/czli7xjp

```
loss real: 0.693, acc real: 0.562 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.690, acc gan: 1.000
loss real: 0.691, acc real: 1.000 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.689, acc gan: 1.000
loss real: 0.689, acc real: 1.000 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.685, acc gan: 1.000
loss real: 0.689, acc real: 1.000 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.686, acc gan: 1.000
loss real: 0.688, acc real: 1.000 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.684, acc gan: 1.000
loss real: 0.687, acc real: 1.000 | loss fake: 0.696, acc fake: 0.000 | loss gan: 0.682, acc gan: 1.000
loss real: 0.686, acc real: 1.000 | loss fake: 0.698, acc fake: 0.000 | loss gan: 0.680, acc gan: 1.000
loss real: 0.685, acc real: 1.000 | loss fake: 0.697, acc fake: 0.000 | loss gan: 0.676, acc gan: 1.000
loss real: 0.685, acc real: 1.000 | loss fake: 0.698, acc fake: 0.000 | loss gan: 0.679, acc gan: 1.000
loss real: 0.684, acc real: 1.000 | loss fake: 0.698, acc fake: 0.000 | loss gan: 0.677, acc gan: 1.000
loss real: 0.682, acc real: 1.000 | loss fake: 0.700, acc fake: 0.000 | loss gan: 0.672, acc gan: 1.000
loss real: 0.681, acc real: 1.000 | loss fake: 0.700, acc fake: 0.000 | loss gan: 0.669, acc gan: 1.000
loss real: 0.680, acc real: 1.000 | loss fake: 0.702, acc fake: 0.000 | loss gan: 0.669, acc gan: 1.000
loss real: 0.683, acc real: 0.812 | loss fake: 0.702, acc fake: 0.000 | loss gan: 0.673, acc gan: 1.000
loss real: 0.685, acc real: 0.781 | loss fake: 0.702, acc fake: 0.000 | loss gan: 0.678, acc gan: 0.969
loss real: 0.683, acc real: 0.812 | loss fake: 0.705, acc fake: 0.000 | loss gan: 0.674, acc gan: 0.844
loss real: 0.682, acc real: 0.812 | loss fake: 0.709, acc fake: 0.000 | loss gan: 0.674, acc gan: 0.906
loss real: 0.682, acc real: 0.750 | loss fake: 0.710, acc fake: 0.000 | loss gan: 0.677, acc gan: 0.750
loss real: 0.685, acc real: 0.781 | loss fake: 0.705, acc fake: 0.062 | loss gan: 0.682, acc gan: 0.688
loss real: 0.686, acc real: 0.656 | loss fake: 0.704, acc fake: 0.094 | loss gan: 0.683, acc gan: 0.688
loss real: 0.686, acc real: 0.594 | loss fake: 0.703, acc fake: 0.062 | loss gan: 0.685, acc gan: 0.656
loss real: 0.684, acc real: 0.844 | loss fake: 0.706, acc fake: 0.031 | loss gan: 0.684, acc gan: 0.812
loss real: 0.684, acc real: 0.750 | loss fake: 0.703, acc fake: 0.031 | loss gan: 0.687, acc gan: 0.781
loss real: 0.688, acc real: 0.656 | loss fake: 0.704, acc fake: 0.000 | loss gan: 0.689, acc gan: 0.656
loss real: 0.686, acc real: 0.812 | loss fake: 0.705, acc fake: 0.094 | loss gan: 0.685, acc gan: 0.625
loss real: 0.687, acc real: 0.688 | loss fake: 0.702, acc fake: 0.125 | loss gan: 0.687, acc gan: 0.438
loss real: 0.690, acc real: 0.531 | loss fake: 0.699, acc fake: 0.562 | loss gan: 0.689, acc gan: 0.281
loss real: 0.691, acc real: 0.469 | loss fake: 0.699, acc fake: 0.656 | loss gan: 0.690, acc gan: 0.312
loss real: 0.689, acc real: 0.562 | loss fake: 0.696, acc fake: 0.625 | loss gan: 0.692, acc gan: 0.281
loss real: 0.689, acc real: 0.594 | loss fake: 0.695, acc fake: 0.594 | loss gan: 0.694, acc gan: 0.312
loss real: 0.688, acc real: 0.656 | loss fake: 0.691, acc fake: 0.656 | loss gan: 0.699, acc gan: 0.219
loss real: 0.691, acc real: 0.344 | loss fake: 0.696, acc fake: 0.281 | loss gan: 0.693, acc gan: 0.406
loss real: 0.691, acc real: 0.375 | loss fake: 0.693, acc fake: 0.344 | loss gan: 0.697, acc gan: 0.406
loss real: 0.692, acc real: 0.357 | loss fake: 0.687, acc fake: 0.500 | loss gan: 0.702, acc gan: 0.071
loss real: 0.691, acc real: 0.406 | loss fake: 0.686, acc fake: 0.594 | loss gan: 0.706, acc gan: 0.375
loss real: 0.692, acc real: 0.438 | loss fake: 0.684, acc fake: 0.594 | loss gan: 0.707, acc gan: 0.281
loss real: 0.692, acc real: 0.344 | loss fake: 0.684, acc fake: 0.531 | loss gan: 0.704, acc gan: 0.281
loss real: 0.691, acc real: 0.469 | loss fake: 0.676, acc fake: 0.562 | loss gan: 0.718, acc gan: 0.406
loss real: 0.692, acc real: 0.438 | loss fake: 0.672, acc fake: 0.469 | loss gan: 0.725, acc gan: 0.375
loss real: 0.691, acc real: 0.688 | loss fake: 0.662, acc fake: 0.406 | loss gan: 0.736, acc gan: 0.281
loss real: 0.698, acc real: 0.406 | loss fake: 0.669, acc fake: 0.312 | loss gan: 0.729, acc gan: 0.469
loss real: 0.693, acc real: 0.562 | loss fake: 0.654, acc fake: 0.406 | loss gan: 0.749, acc gan: 0.344
loss real: 0.694, acc real: 0.438 | loss fake: 0.658, acc fake: 0.375 | loss gan: 0.746, acc gan: 0.344
loss real: 0.694, acc real: 0.531 | loss fake: 0.657, acc fake: 0.312 | loss gan: 0.745, acc gan: 0.281
loss real: 0.685, acc real: 0.781 | loss fake: 0.627, acc fake: 0.625 | loss gan: 0.800, acc gan: 0.344
loss real: 0.686, acc real: 0.719 | loss fake: 0.612, acc fake: 0.625 | loss gan: 0.826, acc gan: 0.344
loss real: 0.699, acc real: 0.562 | loss fake: 0.626, acc fake: 0.656 | loss gan: 0.795, acc gan: 0.344
loss real: 0.692, acc real: 0.594 | loss fake: 0.624, acc fake: 0.406 | loss gan: 0.811, acc gan: 0.344
loss real: 0.680, acc real: 0.688 | loss fake: 0.628, acc fake: 0.688 | loss gan: 0.812, acc gan: 0.281
loss real: 0.679, acc real: 0.688 | loss fake: 0.607, acc fake: 0.719 | loss gan: 0.838, acc gan: 0.281
loss real: 0.675, acc real: 0.688 | loss fake: 0.595, acc fake: 0.688 | loss gan: 0.861, acc gan: 0.312
loss real: 0.669, acc real: 0.625 | loss fake: 0.587, acc fake: 0.625 | loss gan: 0.891, acc gan: 0.344
loss real: 0.664, acc real: 0.625 | loss fake: 0.570, acc fake: 0.625 | loss gan: 0.932, acc gan: 0.344
loss real: 0.658, acc real: 0.562 | loss fake: 0.574, acc fake: 0.562 | loss gan: 0.929, acc gan: 0.438
loss real: 0.641, acc real: 0.656 | loss fake: 0.522, acc fake: 0.625 | loss gan: 1.044, acc gan: 0.219
loss real: 0.617, acc real: 0.781 | loss fake: 0.468, acc fake: 0.719 | loss gan: 1.183, acc gan: 0.000
loss real: 0.625, acc real: 0.594 | loss fake: 0.658, acc fake: 0.344 | loss gan: 0.855, acc gan: 0.094
loss real: 0.601, acc real: 1.000 | loss fake: 0.638, acc fake: 0.719 | loss gan: 0.858, acc gan: 0.000
loss real: 0.594, acc real: 0.750 | loss fake: 0.676, acc fake: 0.875 | loss gan: 0.817, acc gan: 0.031
loss real: 0.580, acc real: 0.781 | loss fake: 0.717, acc fake: 0.719 | loss gan: 0.755, acc gan: 0.281
loss real: 0.600, acc real: 0.781 | loss fake: 0.709, acc fake: 0.719 | loss gan: 0.767, acc gan: 0.281
loss real: 0.533, acc real: 0.969 | loss fake: 0.747, acc fake: 0.562 | loss gan: 0.716, acc gan: 0.469
loss real: 0.523, acc real: 0.969 | loss fake: 0.914, acc fake: 0.312 | loss gan: 0.526, acc gan: 0.812
loss real: 0.531, acc real: 1.000 | loss fake: 0.940, acc fake: 0.281 | loss gan: 0.602, acc gan: 0.719
loss real: 0.583, acc real: 0.938 | loss fake: 0.820, acc fake: 0.438 | loss gan: 0.682, acc gan: 0.562
loss real: 0.552, acc real: 1.000 | loss fake: 0.888, acc fake: 0.250 | loss gan: 0.611, acc gan: 0.750
loss real: 0.576, acc real: 0.969 | loss fake: 0.806, acc fake: 0.375 | loss gan: 0.686, acc gan: 0.625
loss real: 0.581, acc real: 1.000 | loss fake: 0.783, acc fake: 0.375 | loss gan: 0.706, acc gan: 0.625
loss real: 0.585, acc real: 1.000 | loss fake: 0.715, acc fake: 0.469 | loss gan: 0.769, acc gan: 0.531
loss real: 0.584, acc real: 1.000 | loss fake: 0.755, acc fake: 0.375 | loss gan: 0.729, acc gan: 0.625
loss real: 0.582, acc real: 1.000 | loss fake: 0.760, acc fake: 0.344 | loss gan: 0.734, acc gan: 0.656
loss real: 0.586, acc real: 0.969 | loss fake: 0.754, acc fake: 0.281 | loss gan: 0.710, acc gan: 0.719
loss real: 0.593, acc real: 0.969 | loss fake: 0.735, acc fake: 0.344 | loss gan: 0.756, acc gan: 0.656
loss real: 0.594, acc real: 1.000 | loss fake: 0.663, acc fake: 0.438 | loss gan: 0.853, acc gan: 0.562
loss real: 0.593, acc real: 1.000 | loss fake: 0.715, acc fake: 0.312 | loss gan: 0.781, acc gan: 0.688
loss real: 0.613, acc real: 1.000 | loss fake: 0.538, acc fake: 0.643 | loss gan: 0.962, acc gan: 0.500
loss real: 0.590, acc real: 1.000 | loss fake: 0.630, acc fake: 0.438 | loss gan: 0.927, acc gan: 0.562
loss real: 0.578, acc real: 1.000 | loss fake: 0.737, acc fake: 0.250 | loss gan: 0.780, acc gan: 0.750
loss real: 0.591, acc real: 0.969 | loss fake: 0.684, acc fake: 0.312 | loss gan: 0.854, acc gan: 0.688
loss real: 0.600, acc real: 0.969 | loss fake: 0.591, acc fake: 0.469 | loss gan: 0.957, acc gan: 0.531
loss real: 0.584, acc real: 1.000 | loss fake: 0.755, acc fake: 0.219 | loss gan: 0.895, acc gan: 0.656
loss real: 0.591, acc real: 1.000 | loss fake: 0.826, acc fake: 0.000 | loss gan: 0.627, acc gan: 0.625
loss real: 0.607, acc real: 0.969 | loss fake: 0.794, acc fake: 0.281 | loss gan: 0.648, acc gan: 0.656
loss real: 0.635, acc real: 0.875 | loss fake: 0.780, acc fake: 0.375 | loss gan: 0.666, acc gan: 0.625
loss real: 0.639, acc real: 0.875 | loss fake: 0.747, acc fake: 0.500 | loss gan: 0.697, acc gan: 0.500
loss real: 0.630, acc real: 0.938 | loss fake: 0.750, acc fake: 0.344 | loss gan: 0.680, acc gan: 0.656
loss real: 0.645, acc real: 0.812 | loss fake: 0.725, acc fake: 0.438 | loss gan: 0.722, acc gan: 0.531
loss real: 0.639, acc real: 0.969 | loss fake: 0.711, acc fake: 0.438 | loss gan: 0.733, acc gan: 0.562
loss real: 0.639, acc real: 0.875 | loss fake: 0.726, acc fake: 0.312 | loss gan: 0.697, acc gan: 0.688
loss real: 0.646, acc real: 0.781 | loss fake: 0.733, acc fake: 0.250 | loss gan: 0.689, acc gan: 0.750
loss real: 0.639, acc real: 0.875 | loss fake: 0.694, acc fake: 0.406 | loss gan: 0.742, acc gan: 0.594
loss real: 0.660, acc real: 0.812 | loss fake: 0.684, acc fake: 0.438 | loss gan: 0.760, acc gan: 0.562
loss real: 0.634, acc real: 0.938 | loss fake: 0.700, acc fake: 0.312 | loss gan: 0.734, acc gan: 0.688
loss real: 0.639, acc real: 0.938 | loss fake: 0.705, acc fake: 0.250 | loss gan: 0.726, acc gan: 0.750
loss real: 0.657, acc real: 0.812 | loss fake: 0.681, acc fake: 0.312 | loss gan: 0.754, acc gan: 0.688
```

```
loss real: 0.651, acc real: 0.875 | loss fake: 0.628, acc fake: 0.469 | loss gan: 0.830, acc gan: 0.531
loss real: 0.653, acc real: 0.938 | loss fake: 0.651, acc fake: 0.375 | loss gan: 0.802, acc gan: 0.625
loss real: 0.645, acc real: 0.875 | loss fake: 0.627, acc fake: 0.406 | loss gan: 0.833, acc gan: 0.594
loss real: 0.634, acc real: 0.969 | loss fake: 0.674, acc fake: 0.250 | loss gan: 0.772, acc gan: 0.750
loss real: 0.682, acc real: 0.688 | loss fake: 0.599, acc fake: 0.469 | loss gan: 0.892, acc gan: 0.531
```

Waiting for W&B process to finish... **(success).**

```
VBox(children=(Label(value='0.001 MB of 0.025 MB uploaded (0.000 MB deduped)\r'), FloatProgress(value=0.052068…
```

Run history:

acc_fake	
acc_gan	
acc_real	
loss_fake	
loss_gan	
loss_real	

Run summary:

acc_fake	0.46875
acc_gan	0.53125
acc_real	0.6875
loss_fake	0.5989
loss_gan	0.89223
loss_real	0.68225

View run **ritualistic-possession-1** at: https://wandb.ai/leozhao/PetFinder-GANs/runs/czli7xjp

Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `./wandb/run-20231030_213336-czli7xjp/logs`

In [54]:

```
def generate_samples(class_for, n_samples=20):
    '''
    Generates new random but very realistic features using
    a trained generator model

    Params:
        class_for: Int - features for this class
        n_samples: Int - how many samples to generate
    '''

    noise = np.random.uniform(0, 1, (n_samples, latent_dim))
    label = np.full((n_samples,), fill_value=class_for)
    return generator.predict([noise, label])
```

In [55]:

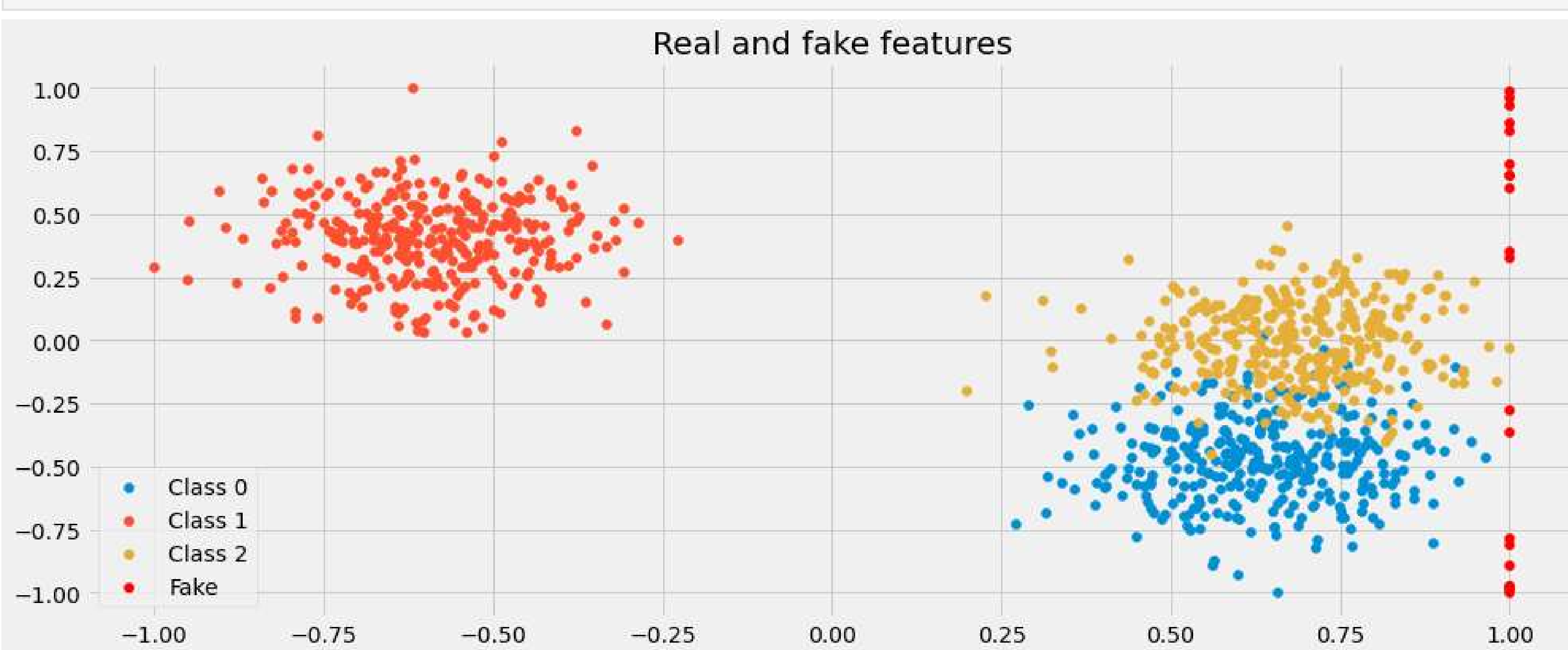
```
features_class_0 = generate_samples(0)
```

In [56]:

```
def visualize_fake_features(fake_features, figsize=(15, 6), color='r'):
    ax, fig = plt.subplots(figsize=figsize)

    # Let's plot our dataset to compare
    for i in range(n_classes):
        plt.scatter(scaled_X[:, 0][np.where(y==i)], scaled_X[:, 1][np.where(y==i)])

    plt.scatter(fake_features[:, 0], fake_features[:, 1], c=color)
    plt.title('Real and fake features')
    plt.legend(['Class 0', 'Class 1', 'Class 2', 'Fake'])
```

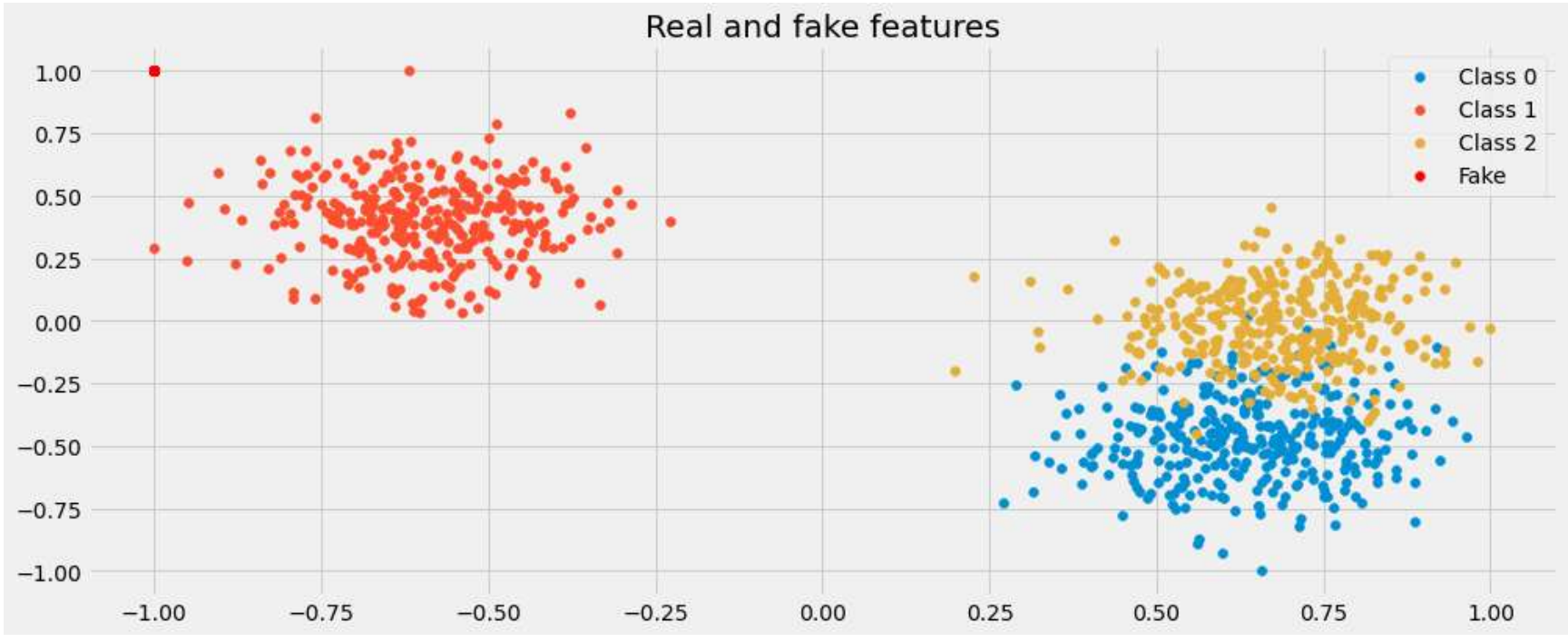
In [57]:

```
visualize_fake_features(features_class_0)
```



In [58]:

```
features_class_1 = generate_samples(1)
visualize_fake_features(features_class_1)
```


Real and fake features
Class 0
Class 1
Class 2
Fake
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00