EasyCV

Release 0.3.0

Oct 21, 2022

Contents

Computer Vision made easy

CHAPTER 1

Installation

You can install **EasyCV** with `pip` or from source.

Note: **EasyCV** only supports Python 3 so make you sure you have it installed before proceeding.

1.1 Using Pip

First, ensure that you have the latest pip version to avoid dependency errors:

```
pip install --upgrade pip
```

Then install **EasyCV** and all its dependencies using pip:

```
pip install easycv
```

1.2 Install from Source

To install EasyCV from source, clone the repository from github:

```
git clone https://github.com/Resi-Coders/easycv.git
cd easycv
pip install .
```

You can view the list of all dependencies within the `install_requires` field of `setup.py`.

1.3 Run Tests

Test EasyCV with `pytest`. If you don't have `pytest` installed run:

```
pip install pytest
```

Then to run all tests just run:

```
cd easycv
pytest .
```

CHAPTER 2

Examples

2.1 Transforms examples

2.1.1 Color Examples

[1]:
```
from easycv import Image, Pipeline
from easycv.transforms import GrayScale, Sepia, FilterChannels, GammaCorrection,␣
↪Negative, Cartoon, PhotoSketch, ColorTransfer, Colorize, ColorPick, Quantitization,␣
↪Hue, Contrast, Brightness, Hsv
```

For this example we will load an image with random(). For more examples of image loading check *Image examples*.

[2]:
```
img = Image.random()
img
```

[2]:

Gray Scale

To convert the image to grayscale we just need to apply the transform, for more information check the reference.

[3]:
```
img.apply(GrayScale())
```

[3]:

Sepia

To convert the image to grayscale we just need to apply the transform, for more information check the reference.

[4]:
```
img.apply(Sepia())
```

[4]:

Filter Channels

To remove a channel we just need to apply the transform with a list of the channels to remove, for more information check the reference.

[5]:
```
img.apply(FilterChannels(channels=[1]))
```

[5]:

Gamma Correction

To apply gamma correction we just need to apply the transform with a new gamma value, for more information check the reference.

[6]:
```
img.apply(GammaCorrection(gamma=3.0))
```

[6]:

[7]:
```
img.apply(GammaCorrection(gamma=0.5))
```

[7]:

Negative

To convert the image to negative we just need to apply the transform, for more information check the reference.

[8]:
```
img.apply(Negative())
```

[8]:

Cartoon

To make an image look like a Cartoon we just need to apply the transform, for more information check the reference.

[9]:
```
img.apply(Cartoon())
```

[9]:

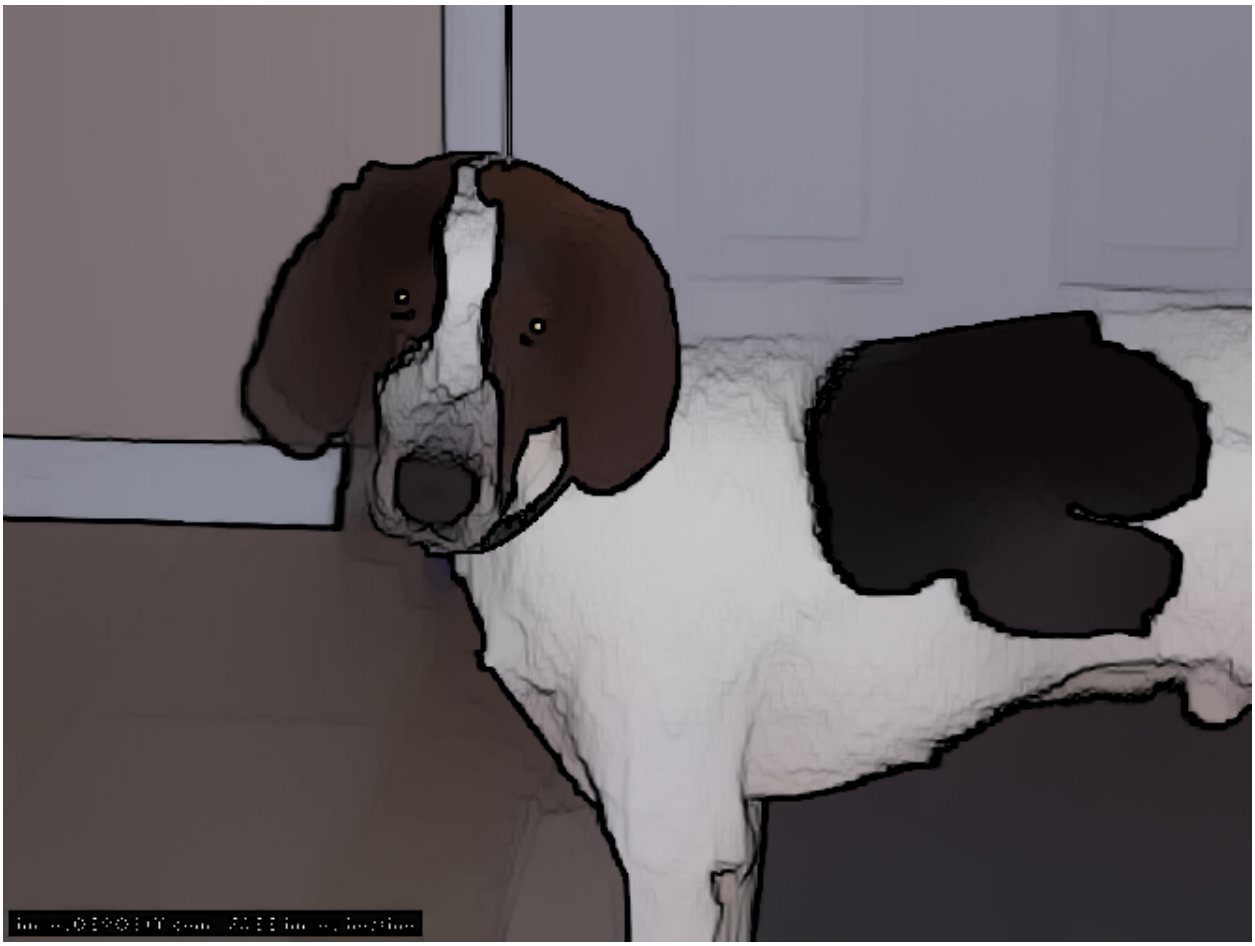
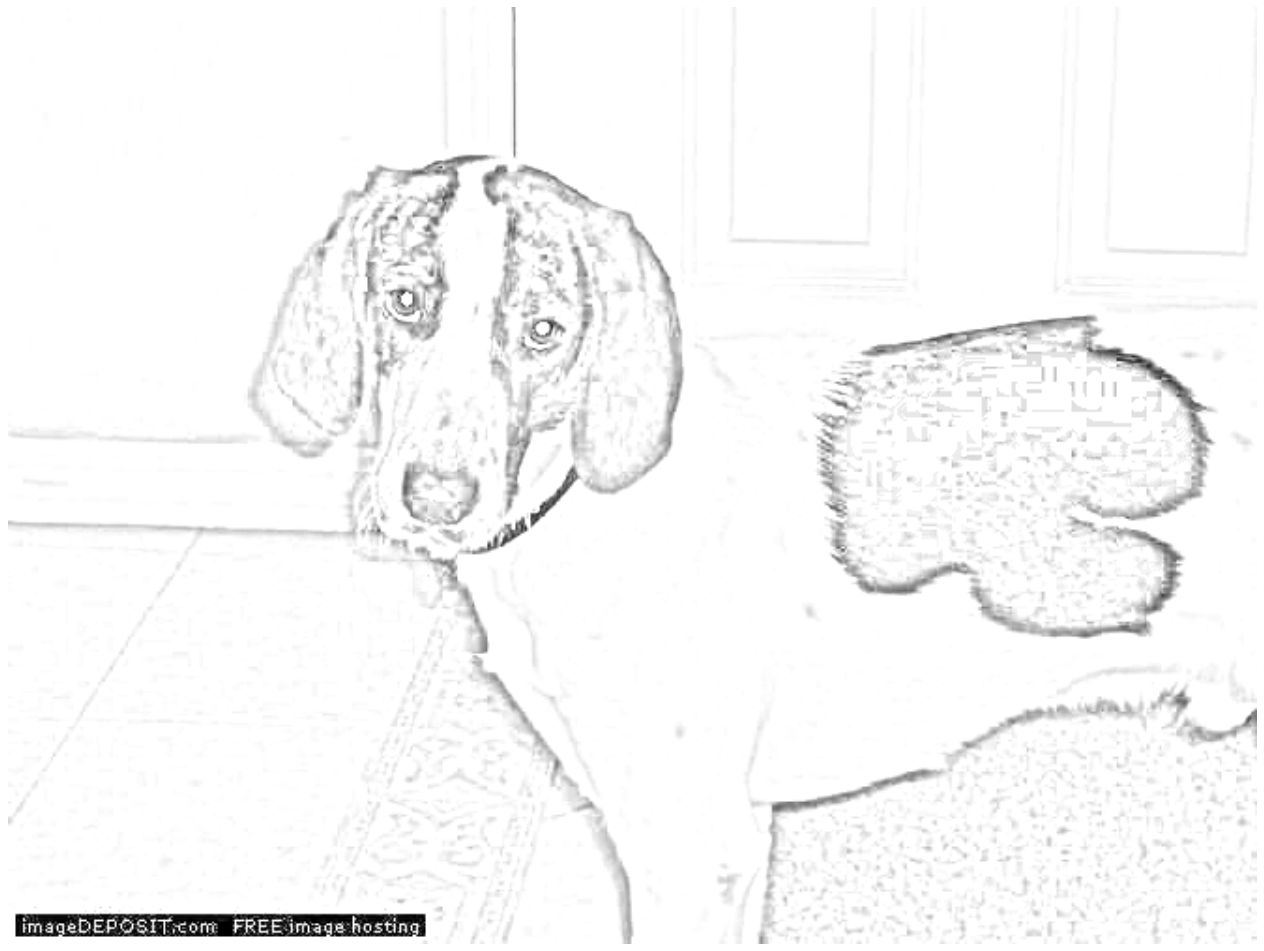


Photo Sketch

To make an image look like a pencil sketch we just need to apply the transform, for more information check the reference.

[10]:
```
img.apply(PhotoSketch())
```

[10]:



Color Transfer

To transfer the colors from source to target we just need to apply the transform, for more information check the reference.

[11]:
```
src = Image("images/source.jpg")
src
```

[11]:

[12]:
```
img.apply(ColorTransfer(source=src))
```

[12]:

Color Pick

To pick the color from the desired point or rectangle we just need to apply the transform, for more information check the reference.

[13]:
```
img.apply(ColorPick(method="point"))
```

[13]:
```
{'color': [127, 116, 120]}
```

Colorize

To colorize an image apply the Colorize transform to a grayscale image, for more information check the reference.

[14]:
```
gray = img.apply(GrayScale())
gray
```

[14]:

[15]:
```
gray.apply(Colorize())
```

[15]:

Quantitization

To reduce the number of colors of an image apply the Quantitization with the number of colors, for more information check the reference.

[16]:
```
img.apply(Quantitization(clusters=8))
```

[16]:

Hue

To add hue to an image apply the Hue with the value to add, for more information check the reference.

[17]:
```
img.apply(Hue(value=10))
```

[17]:

Contrast

To add contrast to an image apply the Contrast with alpha value of contrast to add, for more information check the reference.

[18]:
```
img.apply(Contrast(alpha=1.4))
```

[18]:

Brightness

To add brightness to an image apply the Brightness brightness to add, for more information check the reference.

[19]:
```
img.apply(Brightness(beta=10))
```

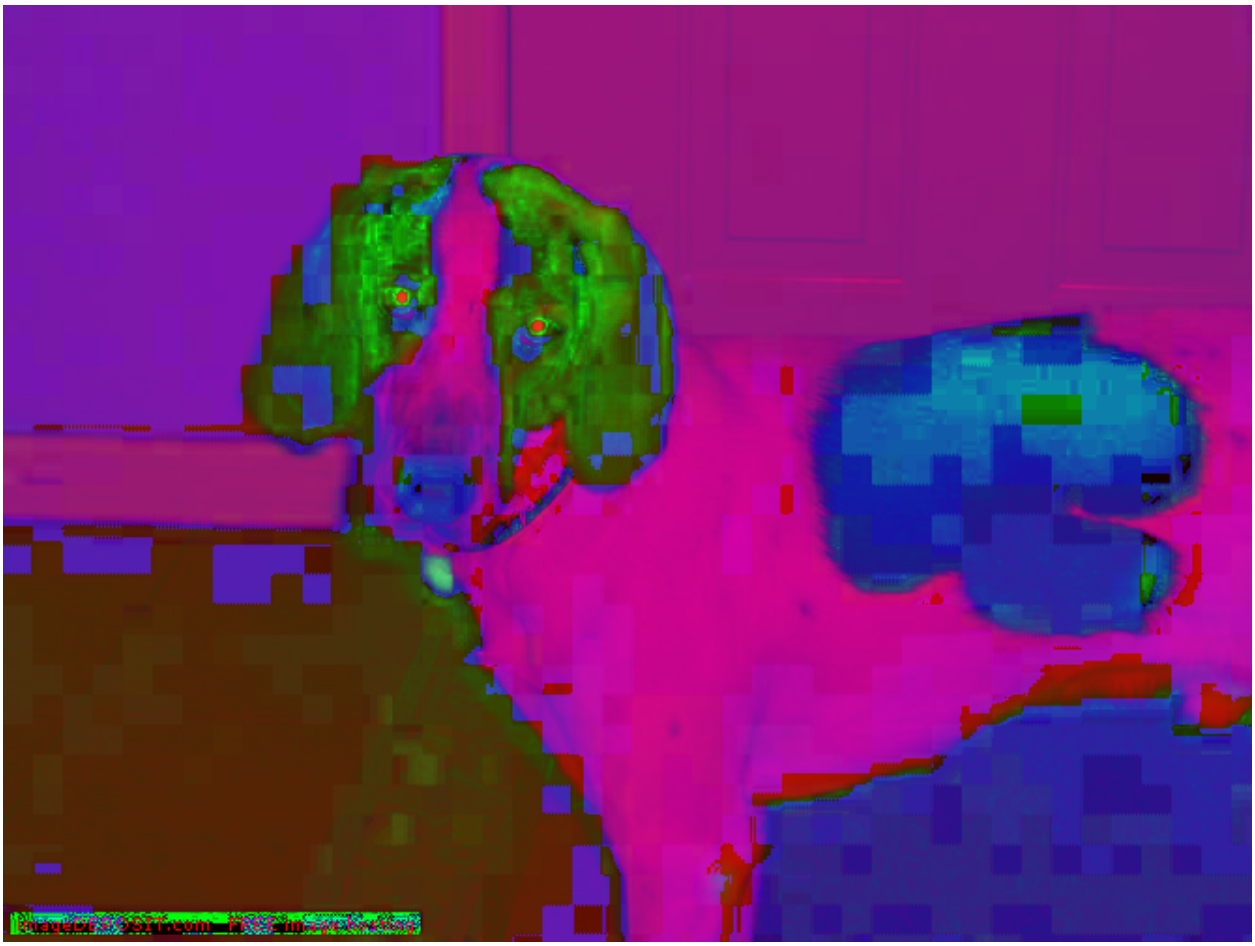
[19]:

Hsv

To reduce the number of colors of an image apply the Quantitization with the number of colors, for more information check the reference.

[20]:
```
img.apply(Hsv())
```

[20]:



2.1.2 Detect Examples

[1]:
```
from easycv import Image
from easycv.transforms import Scan, Lines, Circles
```

For this example we will load an example image from the images folder.

[2]:
```
img = Image("images/qr.png")
img
```

[2]:

Scan

We can obtain all qrcodes and barcodes in the image by calling the Scan transform. The transform returns the number of detections, the data encoded on each one and the bounding boxes. For more information check the reference.

[3]:
```
img.apply(Scan())
```

[3]:
```
{'detections': 2,
 'data': ['https://github.com/easycv/easycv', '12345670'],
 'rectangles': [[(122, 166), (228, 259)], [(304, 225), (413, 249)]]}
```

Lines

To get the lines in an image, call the Lines Transform, for more information check the reference.

[4]:
```
img.apply(Lines(threshold=100))
```

[4]:
```
{'lines': [[[-817, 720], [1084, 102]],
  [[-803, 736], [1087, 85]],
```

(continues on next page)

(continued from previous page)

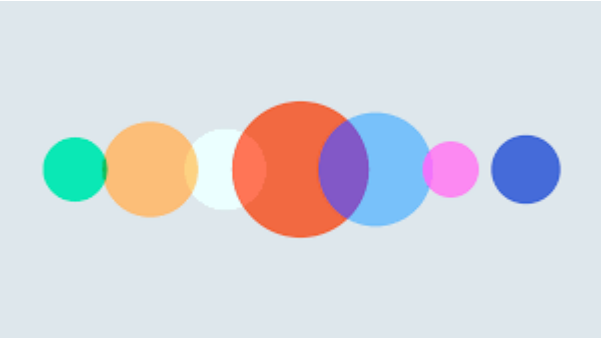
```
  [[-214, -976], [235, 971]],
  [[-212, -1025], [756, 723]],
  [[-818, 717], [1083, 99]],
  [[-929, 403], [1010, -79]],
  [[-235, -1018], [764, 713]],
  [[-260, -1010], [769, 704]]]}
```

Circles

To find the circles in the image, call the Circles transform, for more information check the reference.

[5]:
```
img = Image("images/circles.png")
img
```

[5]:



[6]:
```
img.apply(Circles(threshold=50))
```

[6]:
```
{'circles': array([[149.5,  83.5,  34.3],
        [ 74.5,  83.5,  23.4],
        [262.5,  83.5,  16.6],
        [ 37.5,  83.5,  15.4],
        [224.5,  83.5,  14. ]], dtype=float32)}
```

2.1.3 Draw Examples

[1]:
```
from easycv import Image
from easycv.transforms.draw import Draw
```

For this example we will load an image with random(). For more examples of image loading check *Image examples*.

[2]:
```
img = Image.random()
img
```

[2]:

Ellipse

To draw an ellipse apply the method Draw and select the method ellipse, for more information check reference.

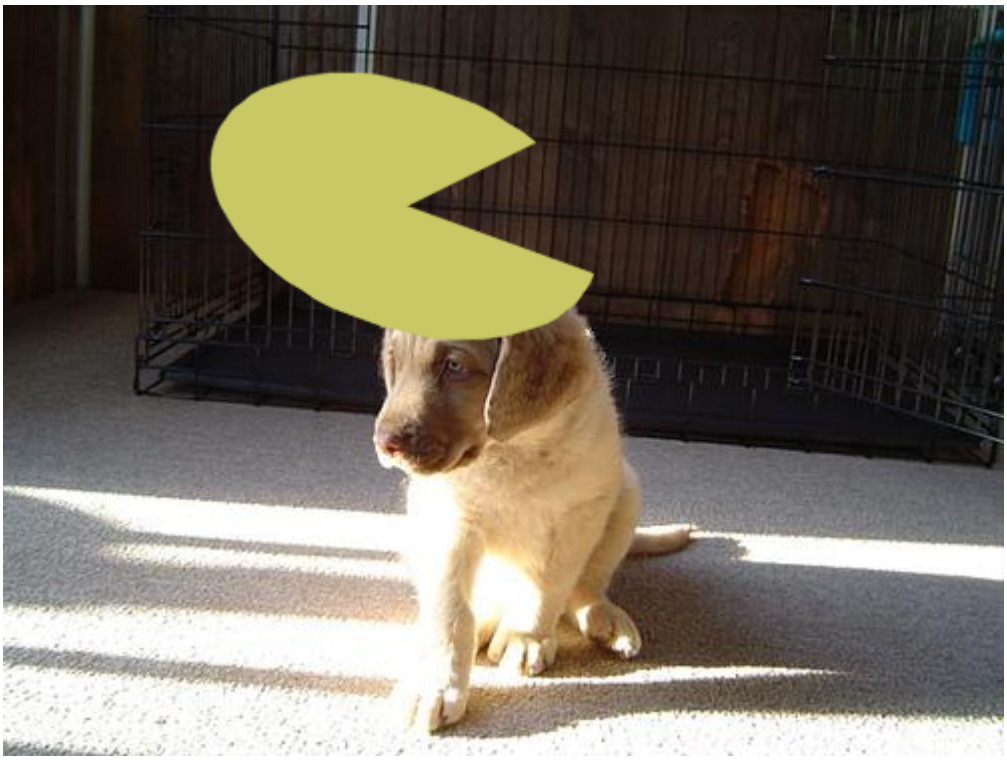
[3]:
```
img.apply(Draw(method="ellipse",
                ellipse=((200,100),100,60),
                rotation_angle=20,
                start_angle=0,
                end_angle=300,
                filled=False,
                color=(100,200,200),
                thickness=5))
```

[3]:

A closed ellipse can also be drawn passing filled as True

[4]:
```
img.apply(Draw(method="ellipse",
               ellipse=((200,100),100,60),
               rotation_angle=20,
               start_angle=0,
               end_angle=300,
               filled=True,
               color=(100,200,200),
               thickness=5,))
```

[4]:



Line

To draw a line apply the method Draw and select the method line, for more information check reference.

[5]:
```
img.apply(Draw(method="line",
                pt1=(30,20),
                pt2=(200,150),
                color=(0,200,0),
                thickness=5))
```

[5]:

Polygon

To draw multiple lines or a polygon apply the method Draw and select the method polygon, for more information check reference.

Not closed and not Filled

[6]:
```
img.apply(Draw(method = "polygon",
               points=[[50,50],[100,100],[50,100],[100,50]],
               color=(250,0,0),
               thickness=2,
               closed=False,
               filled=False))
```

[6]:

Closed and not Filled

[7]:
```
img.apply(Draw(method = "polygon",
               points=[[50,50],[100,100],[50,100],[100,50]],
               color=(0,0,250),
               thickness=2,
               closed=True,
               filled=False))
```

[7]:

Closed and Filled

[8]:
```
img.apply(Draw(method = "polygon",
               points=[[50,50],[100,100],[50,100],[100,50]],
               color=(100,0,70),
               thickness=2,
               closed=False,
               filled=True))
```

[8]:

Rectangle

To draw a rectangle apply the method Draw and select the method rectangle, for more information check reference.

Not Filled

[9]:
```
img.apply(Draw(method="rectangle", rectangle=((95,50),(350,250)),filled=True))
```

[9]:

Filled

[10]:
```
img.apply(Draw(method="rectangle", rectangle=((95,50),(350,250)),color=(50,100,200),
↪filled=False))
```

[10]:

Text

To draw Text apply the method Draw and select the method Text, for more information check reference.

[11]:
```
img.apply(Draw(method = "text", text="easyCV", org=(1,200), font="SIMPLEX", size=4,␣
↪color=(0,150,250),thickness=5))
```

[11]:

The text can be mirrored using x_mirror

[12]:
```
img.apply(Draw(method = "text", text="easyCV", org=(1,200), font="SIMPLEX", size=4,␣
↪color=(0,150,250),thickness=5,x_mirror=True))
```

[12]:

2.1.4 Edges Examples

[1]:
```
from easycv import Image
from easycv.io.output import show_grid
from easycv.transforms import Gradient, GradientAngle, Canny
```

For this example we will load an image from the Dog API. For more examples of image loading check *Image examples*.

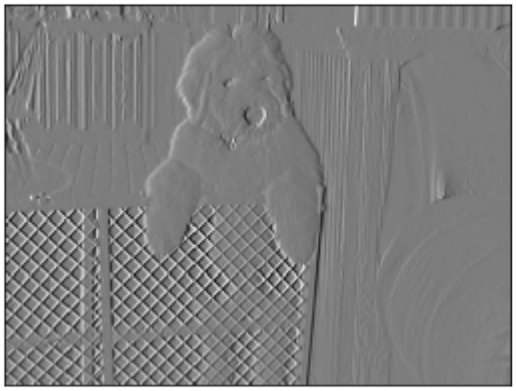
[2]:
```
img = Image("https://images.dog.ceo/breeds/sheepdog-english/n02105641_1045.jpg")
img
```

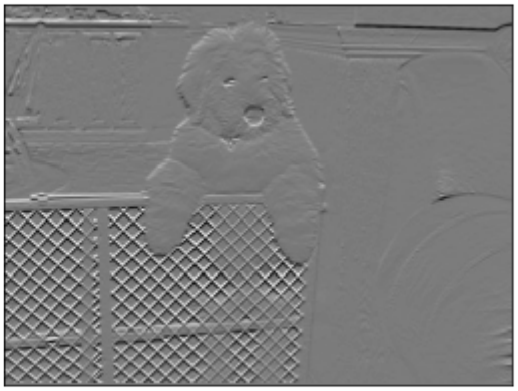
[2]:

Gradient

To calculate the gradient of the image we simply apply the Gradient Transform for both axis, for more information check the reference. The gradients are displayed in a grid created using show_grid().

[3]:
```
grad_x = img.apply(Gradient(axis="x"))
grad_y = img.apply(Gradient(axis="y"))
show_grid([grad_x,grad_y])
```





Gradient Magnitude

To calculate the magnitude of the image gradient we simply apply the Gradient transform with no axis, for more information check the reference.

[4]:
```
img.apply(Gradient())
```

[4]:


Gradient Angle

To calculate the angles of the image gradient we simply apply the GradientAngle transform, for more information check the reference.

[5]:
```
img.apply(GradientAngle(size=1))
```

[5]:

Canny Edge

To get the edges of an image we just apply the Canny transform with the low and high threshold, for more information check the reference.

[6]:
```
img.apply(Canny(low=100, high=200))
```

[6]:

2.1.5 Filter Examples

[1]:
```
from easycv import Image
from easycv.transforms.filter import Blur, Sharpen, Sharpness
```

For this example we will load an image with random(). For more examples of image loading check *Image examples*.

[2]:
```
img = Image.random()
img
```

[2]:

Blur

To blur an image we simply apply the Blur transform with the desired method, for more information check the reference.

[3]:
```
img.apply(Blur(method="gaussian", size=11))
```

[3]:

[4]:
```
img.apply(Blur(method="bilateral"))
```

[4]:

Sharpness

You can detect how sharpen an image image by applying the Sharpness transform, for more information check the reference.

[6]:
```
img
```

[6]:


[5]:
```
img.apply(Sharpness())
```

[5]:
```
{'sharpness': 325.05299530543067, 'sharpen': True}
```

We can see that the image is sharpen. Now let's test on a blurry image.

[7]:
```
img_blurry = Image('images/blurred.jpg')
img_blurry
```

[7]:

[8]:
```
img_blurry.apply(Sharpness())
```

[8]:
```
{'sharpness': 14.315561379722062, 'sharpen': False}
```

Sharpen

To Sharpen an image we simply apply the Sharpen transform, for more information check the reference.

[9]:
```
img.apply(Sharpen())
```

[9]:

2.1.6 Noise Examples

[5]:
```
from easycv import Image
from easycv.transforms import Noise
```

For this example we will load an image with random(). For more examples of image loading check *Image examples*.

[6]:
```
img = Image.random()
img
```

[6]:

Noise

To add noise to an image we simply apply the Noise Transform with the desired method, for more information check the reference.

[7]:
```
img.apply(Noise(method="gaussian"))
```

[7]:

[8]:
```
img.apply(Noise(method="sp"))
```

[8]:


2.1.7 Perspective Examples

[21]:
```
from easycv import Image, Pipeline
from easycv.transforms import Perspective, Select
```

For this example we will load an example image from the images folder.

[15]:
```
img = Image('images/hello.png')
img
```

[15]:

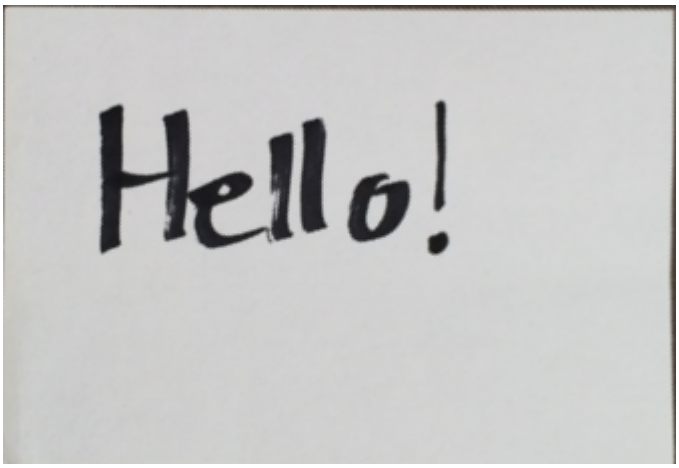
Perspective

To execute a perspective transform we just apply the Prespective Transform with the desired points to the image, for more information check the reference.

[17]:
```
points = [(71, 238), (354, 114), (470, 257), (186, 437)] # corners of the paper

img.apply(Perspective(points=points))
```

[17]:



We could also select the points with the Select Transform, for more information check the reference.

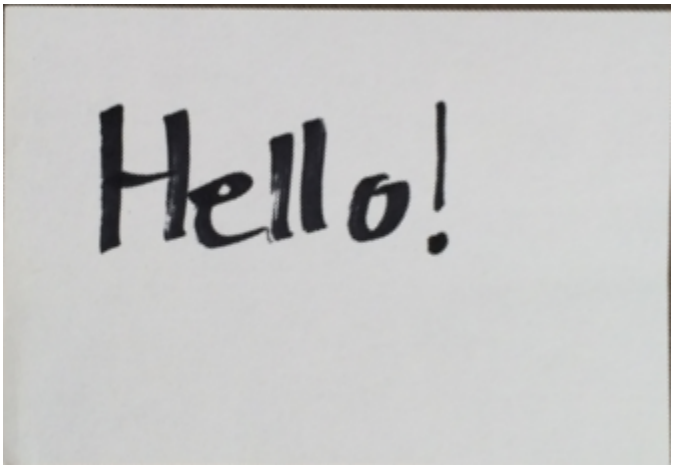
[19]:
```
points = img.apply(Select.point(n=4))["points"]
points
```

[19]:
```
[(71, 238), (352, 114), (470, 258), (187, 437)]
```

And then call as we did in the last example.

[20]:
```
img.apply(Perspective(points=points))
```

[20]:



We can also create a pipeline that selects and applies the perspective transform.

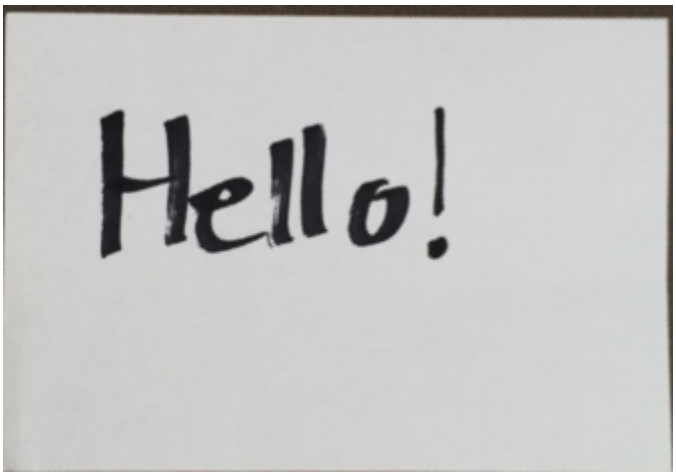
[23]:
```
pipe = Pipeline([Select.point(n=4), Perspective()])
pipe
```

[23]:
```
Pipeline (pipeline) with 2 transforms
    1: Select (method=point, n=4)
    2: Perspective (method=(...), points=(...))
```

This will open a window asking to select the points and then execute a perspective transform on the selected points.

[24]:
```
img.apply(pipe)
```

[24]:



2.1.8 Spatial Examples

[1]:
```
from easycv import Image
from easycv.transforms import Resize, Rescale, Rotate, Crop, Translate, Mirror, Paste
```

For this example we will load an image with random(). For more examples of image loading check *Image examples*.

[2]:
```
img = Image.random()
img
```

[2]:

Resize

To resize an image we simply apply the Rezise Transform, for more information check the reference.

[3]:
```
img.apply(Resize(width=300, height=250))
```

[3]:


Rescale

To rescale an image we simply apply the Rescale Transform with the factors of x and y, for more information check the reference.

[4]:
```
img.apply(Rescale(fx=2, fy=1))
```

[4]:


Rotate

To rotate an image we simply apply the Rotate Transform with the degrees to rotate, for more information check the reference.

[5]:
```
img.apply(Rotate(degrees=50))
```

[5]:

Crop

To crop an image we simply apply the Crop Transform with the boundaries of each side, for more information check the reference.

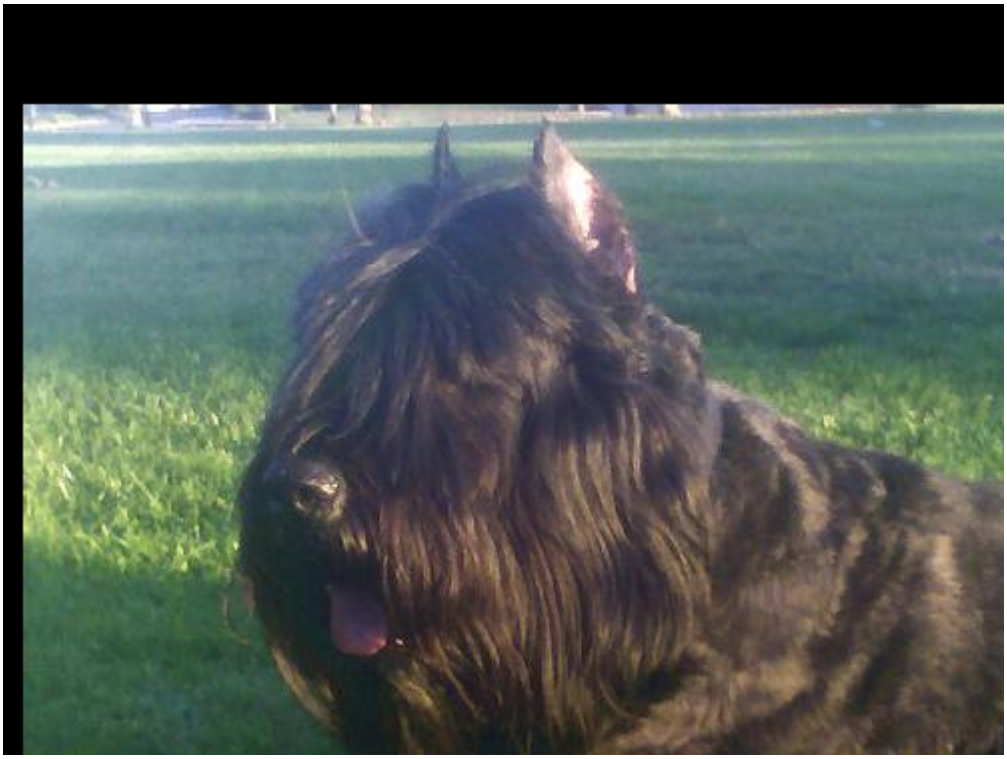
[6]:
```
img.apply(Crop(rectangle=[[50,50],[100,100]]))
```

[6]:

Translate

To translate an image we simply apply the Translate Transform with the vector to translate, for more information check the reference.

[7]:
```
img.apply(Translate(x=10,y=50))
```

[7]:


Mirror

To flip an image we simply apply the Mirror Transform with the desired axis to flip, for more information check the reference.

[8]:
```
img.apply(Mirror(axis="y"))
```

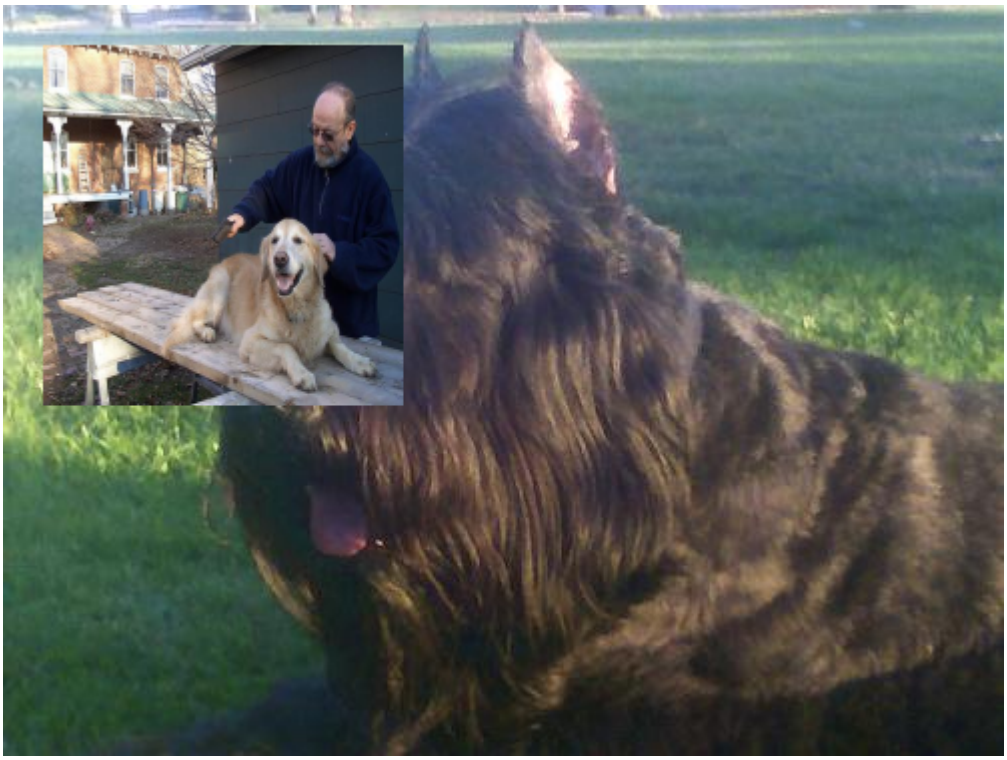
[8]:

Paste

To paste an image on to another simply apply the Paste transform with the image to use and the rectangle to paste, for more information check the reference.

[9]:
```
img.apply(Paste(paste=Image.random(), rectangle=[[20,20],[200,200]]))
```

[9]:



2.1.9 Morphological Examples

[22]:
```
from easycv import Image
from easycv.transforms import Noise, Erode, Dilate, Morphology
```

For this example we will load an example image from the images folder. For more examples of image loading check *Image examples*.

[23]:
```
img = Image("images/logo.png")
img
```

[23]:

Erode

Erosion on an image works like soil erosion, it erodes away the boundaries of objects. For more information check the reference.

[24]:
```
img.apply(Erode())
```

[24]:


Dilate

Dilation is the opposite of erosion. For more information check the reference.

[25]:
```
img.apply(Dilate())
```

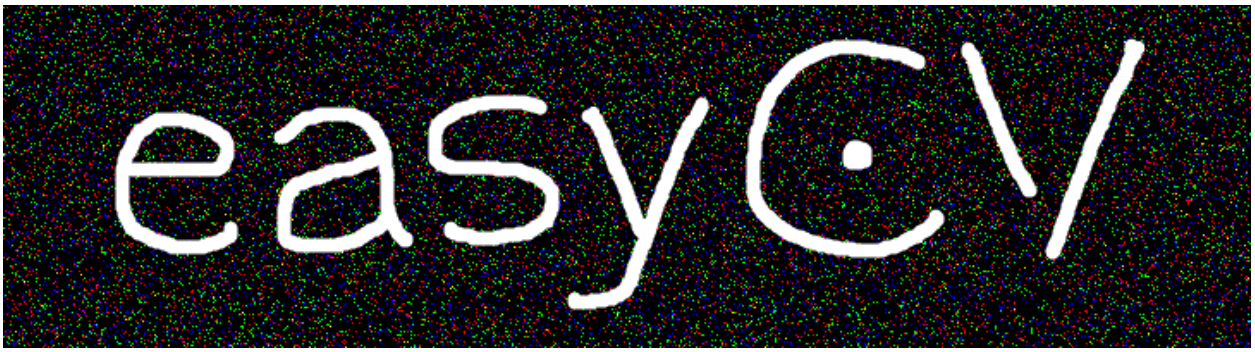
[25]:


Morphology

We can apply different morphological operations through the various methods of Morphology. For more information check the reference.

Let's apply some salt noise to the image

[26]:
```
noisy = img.apply(Noise(method="salt"))
noisy
```

[26]:


Morphology opening method is very good at removing small random noises like this. Let's apply it to clean the image.

[27]:
```
noisy.apply(Morphology(method="opening"))
```

[27]:

Let's apply some pepper noise to the image

[28]:
```
noisy = img.apply(Noise(method="pepper", amount=0.2))
noisy
```

[28]:

Morphology closing method is very good at removing small random noises like this. Let's apply it to clean the image.

[29]:
```
noisy.apply(Morphology(method="closing"))
```

[29]:

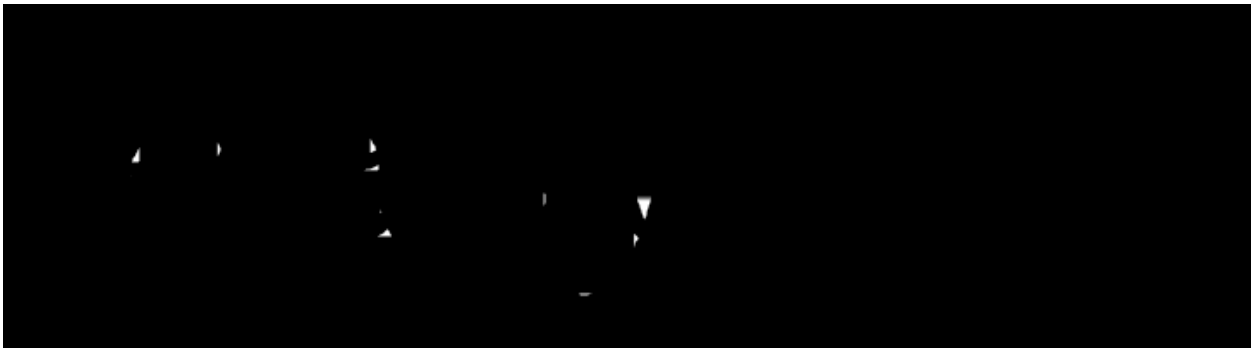
We can also compute things like TopHat (difference between input image and Opening of the image) and BlackHat (difference between the closing of the input image and input image).

[30]:
```
img.apply(Morphology(method="tophat", size=9))
```

[30]:

[31]:
```
img.apply(Morphology(method="blackhat", size=9))
```

[31]:



2.2 Image examples

[1]:
```
from easycv.io import show_grid
from easycv import Image, Pipeline
from easycv.transforms import GrayScale, Gradient, Noise, Blur
```

2.2.1 Loading an image

For this example we will load an image using random().

[2]:
```
img = Image.random()
img
```

[2]:

2.2.2 Image Propeties

We can check images properties such as width and height

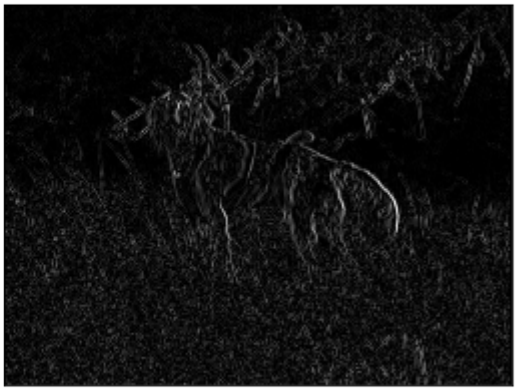
[3]:
```
img.height, img.width
```

[3]:
```
(375, 500)
```

2.2.3 Applying transforms/pipelines

We can apply any transforms to the image

[4]:
```
img_1 = img.apply(GrayScale())
img_2 = img.apply(Gradient())
show_grid([img_1, img_2])
```

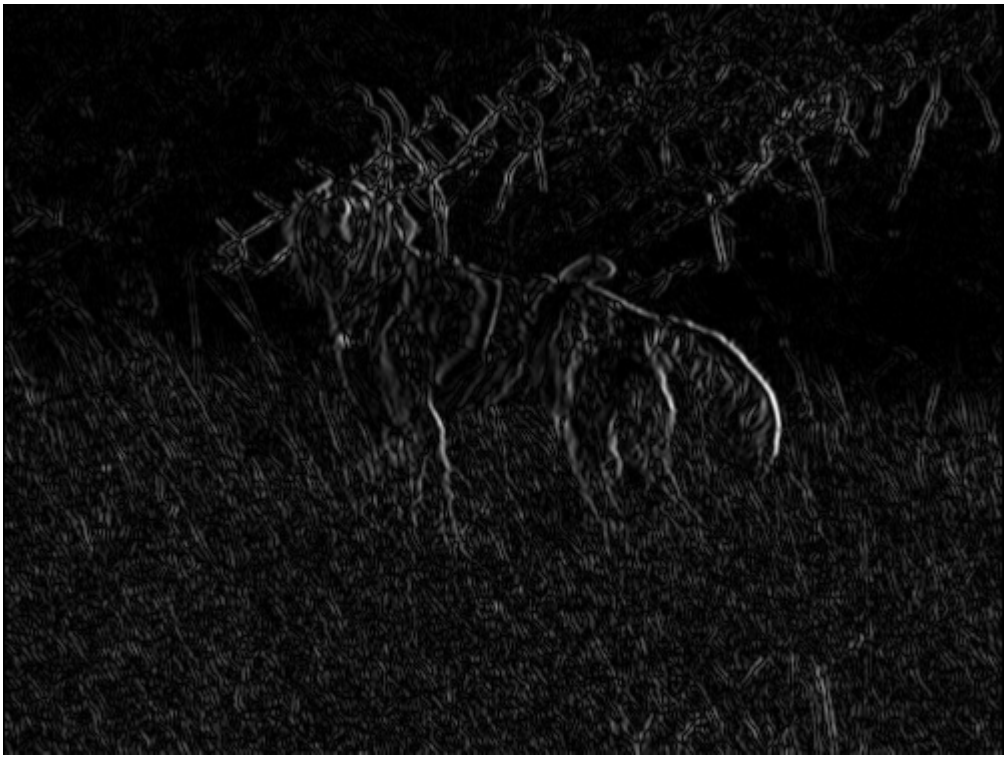


For convenience, images above are displayed in a grid created using *show_grid()*.

We can also apply pipelines the same way

[5]:
```
img_3 = img.apply(Pipeline([Gradient(),Blur()]))
img_3
```

[5]:



2.2.4 Viewing Image

To view the image in a popup window call the show method

[6]:
```
img.show() # Image window pops up
```

2.2.5 Lazy images

Lazy images delay computation while possible. To create a lazy image we just specify the lazy parameter

[7]:
```
img = Image.random()
img.loaded
```

[7]:
```
True
```

Normally transforms/pipelines are executed in the moment. Lazy images will save them for later execution in a pipeline

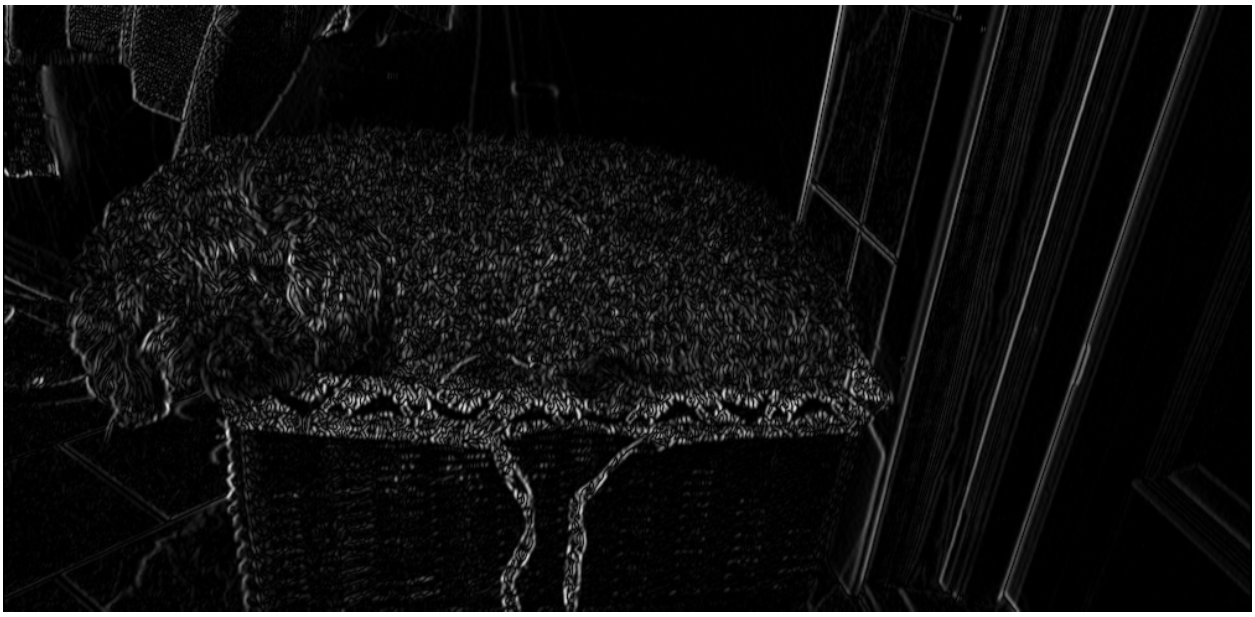
[8]:
```
img = img.apply(Gradient())
img.pending
```

[8]:
```
Pipeline (pending) with 0 transforms
```

When any operation requires the updated image it will be loaded/computed. The changes are saved to prevent duplicated computation. You can force the computation with the compute method but calling any parameter/method (width, height, ...) that requires the updated image would have the same effect. Check the *reference* for more information.

[9]:
```
img.compute()
```

[9]:



[10]:
```
img.loaded
```

[10]:
```
True
```

[11]:
```
img.pending
```

[11]:
```
Pipeline (pending) with 0 transforms
```

2.2.6 Save

In the end we can save the image to a file

[12]:
```
img.save("Image.jpg")
```

2.3 Pipeline examples

[1]:
```
from easycv import Image, Pipeline
from easycv.transforms import Gradient, Blur, Rotate
```

For this example we will load an image using random().

[2]:
```
img = Image.random()
img
```

[2]:

2.3.1 Creating pipeline

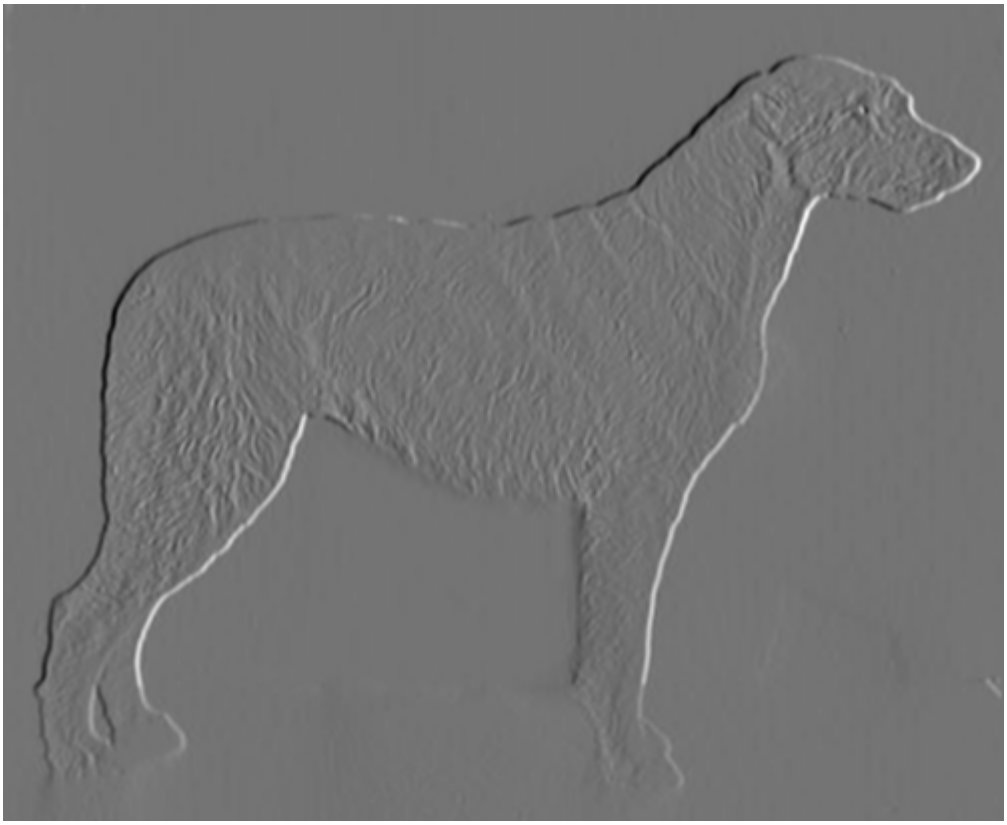
To create a pipeline we simply call it's constructor with a list of transforms.

[3]:
```
pipeline = Pipeline([Gradient(), Blur()], name="test")
pipeline
```

[3]:
```
Pipeline (test) with 2 transforms
    1: Gradient (axis=x, method=sobel, size=5)
    2: Blur (method=gaussian, size=auto, sigma=0, sigma_color=75, sigma_space=75,␣
↪truncate=4)
```

[4]:
```
img.apply(pipeline)
```

[4]:



We can also create a pipeline with another pipeline inside

[5]:
```
pipeline = Pipeline([pipeline, Rotate(degrees=20)],name="test")
pipeline
```

[5]:
```
Pipeline (test) with 3 transforms
    1: Pipeline (test) with 2 transforms
    |    1: Gradient (axis=x, method=sobel, size=5)
    |    2: Blur (method=gaussian, size=auto, sigma=0, sigma_color=75, sigma_space=75,
↪ truncate=4)
    2: Rotate (degrees=20, scale=1, center=auto, original=True)
```

And it's possible to add a transform to an existing pipeline

[6]:
```
pipeline.add_transform(Rotate(degrees=-40))
pipeline
```

[6]:
```
Pipeline (test) with 4 transforms
    1: Pipeline (test) with 2 transforms
    |    1: Gradient (axis=x, method=sobel, size=5)
    |    2: Blur (method=gaussian, size=auto, sigma=0, sigma_color=75, sigma_space=75,
↪ truncate=4)
    2: Rotate (degrees=20, scale=1, center=auto, original=True)
    3: Rotate (degrees=-40, scale=1, center=auto, original=True)
```

2.3.2 Number of transforms

You can check the number of transforms in a pipeline. This is the number of Transforms so nested pipelines count as their internal number of Transforms

[7]:
```
pipeline.num_transforms()
```

[7]:
```
4
```

2.3.3 Clear

You can also remove every transform in a pipeline

[8]:
```
pipeline.clear()
pipeline.num_transforms()
```

[8]:
```
0
```

2.3.4 Save

If you want you can save a pipeline for future use or sharing

[9]:
```
pipeline.save(filename="pipeline")
```

2.4 List Examples

[1]:
```
from easycv import List, Image
from easycv.transforms import GrayScale, Canny
```

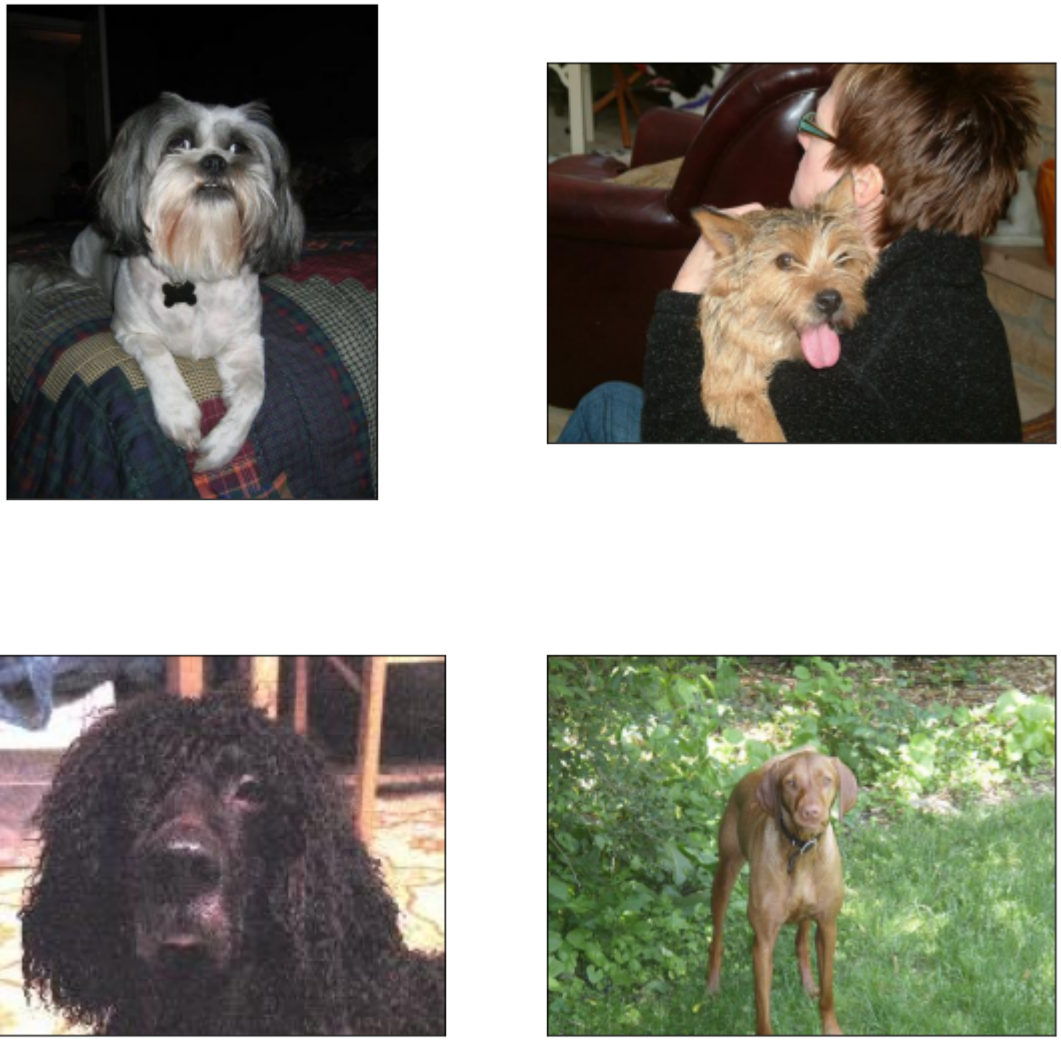
2.4.1 Creating a list

We can create a list of random images using random()

[2]:
```
l = List.random(4)
```

To view the images, we can call the show method

[3]:
```
l.show()
```



2.4.2 Apply transforms

We can apply transforms to List exactly like we would apply them to a single image

[4]:
```
l = l.apply(GrayScale())
l.show()
```




2.4.3 Access single images

Lists support indexing/slicing like normal python lists

[6]:
```
l[1]
```

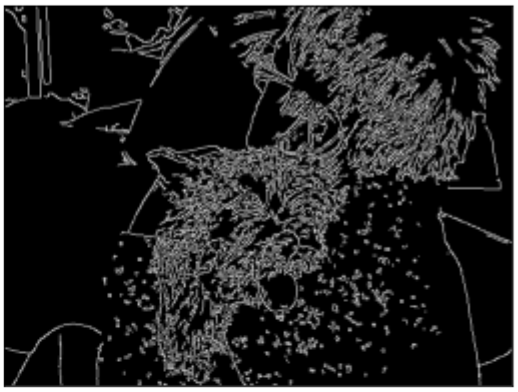
[6]:


[8]:
```
l[2:4].show()
```


2.4.4 Parallel processing

To process images in parallel just set parallel to True on apply. Note that for non-lazy images, small ammounts of images and fast operations this could be slower thatn withou parallelism!

[9]:
```
l = l.apply(Canny(), parallel=True)
l.show()
```



CHAPTER 3

Reference

3.1 Image

The `Image` module provides a class with the same name which is used to represent an **image**.

Images provide a way of loading images easily from multiple sources and a simple but powerful way of applying *Transforms*/*Pipelines*.

> **Warning:** **EasyCV** uses *BGR* color scheme in the internal representation. This makes the interface with opencv easier and faster. Be aware of this if you use the array directly.

3.1.1 Examples

Load Images

The following script loads and displays an **image** from a file and then from a url.

```
from easycv import Image

img = Image("lenna.jpg").show()
img_from_url = Image("www.example.com/lenna.jpg").show()
```

Apply transforms

The following script uses the **image** from the last example. It turns the **image** into grayscale and then blurs it.

```
from easycv.transforms import Blur, Grayscale

img = img.apply(Grayscale())
```

(continues on next page)

(continued from previous page)

```
img = img.apply(Blur(sigma=50))
img.show()
```

Note: If you are running **Easycv** inside a jupyter notebook there is no need to call `show()` , the **image** will be displayed if you evaluate it.

3.1.2 Lazy Images

Lazy images are only loaded/computed when their updated array data is needed. Methods that need the updated image need to have this decorator to ensure that the image is computed before their execution.

3.1.3 Image Class

3.2 Pipeline

The `Pipeline` module provides a class with the same name which is used to represent a Pipeline.

Pipelines can be applied to Images exactly like *Transforms*. They consist of a series of *Transforms* applied in order. They can also be saved/loaded for later use.

3.2.1 Examples

Create simple Pipeline

The following script creates a simple **pipeline** and applies it to a previously loaded **image**. Info on how to load images can be found *here*.

```
from easycv import Pipeline
from easycv.transforms import Blur, Grayscale

pipeline = Pipeline([Grayscale(), Blur(sigma=50)])
img = img.apply(pipeline)
img.show()
```

This is the same as doing the following (applying each **transform** separately):

```
img = img.apply(Grayscale())
img = img.apply(Blur(sigma=50))
img.show()
```

Save and load Pipeline

The following script saves and loads the **pipeline** created in the last example.

```
pipeline.save(filename='example.pipe')

loaded = Pipeline('example.pipe')
```

(continues on next page)

(continued from previous page)

```
img = img.apply(loaded)
img.show()
```

Note: If you are running **Easycv** inside a jupyter notebook there is no need to call *show()* , the image will be displayed if you evaluate it.

3.2.2 Pipeline Class

3.3 List

3.3.1 List Class

3.4 Transforms

Transforms are the most important part of EasyCV. This module contains all the transforms that EasyCV currently supports. It also enables some powerful functionalities that we'll discuss later on this page.

3.4.1 Introduction

All transforms inherit from the base class Transform. The Transform class takes care of most of the core functionality simplifying the process of creating new transforms.

Transforms can be used to modify an image (e.g. blur an image) or to extract valuable information (e.g. count the number of faces in the image). Any of the mentioned types of transform can be applied in the same way. The only difference between them is their outputs.

3.4.2 Transform structure

All transforms follow the same structure. They all inherit from Transform and must override the method `process`. This method receives the image array and should return the result of applying the transform to the image. `process` also receives all the transform arguments, we will talk about arguments in detail later on this documentation.

Let's implement a simple transform that sets the first color channel to zero.

```
from easycv.transforms.base import Transform

class FillChannel(Transform): # inherit from Transform

    def process(self, image, **kwargs): # override process
        image[:, :, 0] = 0
        return image
```

As you can see from the example above it's really simple to extend EasyCV. Now we can use our transform!

```
from easycv import Image

img = Image("lenna.jpg")
img.apply(FillChannel()).show() # Opens a popup window with the altered image
```

Warning: The **process** method always receives the image as a numpy array not as an `easycv.image.Image` instance.

3.4.3 Arguments

Some transforms depend on hyper-parameters or other configurations. EasyCV calls them arguments and they can specified be inside the transform class by assigning a dictionary containing the argument specifications to the variable `arguments`. In this dictionary argument names are the keys and the values are the argument `validators`. Validators enable easy and reliable argument validation/forwarding, more details about validators can be found *here*.

EasyCV will check if the user is inserting the required arguments (arguments without a default value are considered required) and if the inserted values are valid. A helpful error message is generated automatically from the argument validator is generated in case of an error.

Let's add argument to the transform we created above. We'll add the argument `fill_value` to enable people to change the value we use to fill the channel! Since `fill_value` must be an 8-bit integer we'll use a `Number` validator with some restrictions applied.

Arguments are given to `process` through `kwargs`. You can assume that process only runs if all arguments are valid and all required arguments have been filled.

Note: The **process** method receives all the specified arguments on `kwargs` regardless of which arguments the user enters (default values are used for this). There is an exception to this case we'll discuss later (method-specific arguments).

```
from easycv.validators import Number

class FillChannel(Transform):

    # define arguments
    arguments = {
        "fill_value": Number(min_value=0, max_value=255, only_integer=True,␣
↪default=0),
    }

    def process(self, image, **kwargs):
        image[:, :, 0] = kwargs["fill_value"] # use the new argument instead of␣
↪always zero
        return image
```

Now we can use the updated transform with the `fill_value` argument!

```
from easycv import Image

img = Image("lenna.jpg")
img.apply(FillChannel(fill_value=128)).show() # Let's fill the image channel with 128
```

But what happens if the argument is invalid? Let's check!

```
img.apply(FillChannel(fill_value="bad value")).show()

>>> (Exception raised)
>>> InvalidArgumentError: Invalid value for fill_value. Must be an integer between 0␣
↪and 255.
```

As we can see a helpful message is displayed warning the user that `fill_value` must be an integer between 0 and 255.

3.4.4 List of Transforms

Color

The Color module provides transforms that operate on the Image color. Examples of each transform in this module can be found *here*.

Detect

The Detect module provides transforms that detect features/objects on an image. Examples of each transform in this module can be found *here*.

Draw

The Draw transform provides a way to draw 2D shapes or text on a image. *here*.

Edges

The Edges module provides transforms that allow the user to perform edge detection and related tasks. Examples of each transform in this module can be found *here*.

Filter

The Filter module provides transforms that filter an image. Examples of each transform in this module can be found *here*.

Noise

The Noise module provides transforms that add noise to images. Examples of each transform in this module can be found *here*.

Selectors

The selector module allows the user to manually select a shape within an image for other operations

Perspective

The Perspective module provides transforms that alter the perspective of the image. Examples of each transform in the module can be found *here*.

Spatial

The Spatial module provides transforms that alter the spacial properties of the image. Examples of each transform from in module can be found *here*.

Morphological

The Morphological module provides transforms that apply simple operations based on the image shape. Examples of each transform in this module can be found *here*.

3.5 Validators

3.6 Resources

Resources are an easy way to manage large files that transforms need and are not included on easycv by default. Examples of such files are datasets and pre-trained models.

3.6.1 Creating a resource

Resources are represented by a YAML file. You can manually create this file or use the built-in resource creator.

Using resource creator

The following script creates a resource.

```
from easycv.resources import create_resource

create_resource()

>>> Resource name: test-resource
>>> Number of files: 1
>>> File 1 name: file1.txt
>>> File 1 url: example.com/file1.txt
>>> Downloading/hashing files...
>>> Resource created successfully
```

Manually

A resource can be represented by YAML file. We could create the resource described above with the following file:

To add this resource to easycv, save the file as **test-resource.yaml** and put it inside **easycv\resources\sources**.

3.6.2 Using resources

Resources can be used inside transforms to access big files that don't ship with EasyCV by default.

For example, if you want to create a transform that uses a pre-trained neural network you can create a resource that represents the weights file. Then on your transform, if you require the resource, EasyCV downloads the resource automatically when the transform is used.

Let's create a transform called TestTransform that uses the resource we created above. We can obtain a path to the file by using `get_resource`.

```
from easycv.resources import get_resource
from easycv.transforms.base import Transform

class TestTransform(Transform):
    def process(self, image, **kwargs):
        file = get_resource("test-resource", "file1.txt")
        (...) # do something with the file and the image
        return image
```

3.6.3 Functions

3.7 IO

3.7.1 Input

The input module provides functions related to image intake

3.7.2 Output

The output module provides the functions to handle showing and saving images

3.8 Errors

3.8.1 io

3.8.2 transforms

CHAPTER 4

Indices and tables

- genindex
- modindex
- search