search_images_torch

June 14, 2022

1 Search images with deep learning (torch)

Images are usually very different if we compare them at pixel level but that's quite different if we look at them after they were processed by a deep learning model. We convert each image into a feature vector extracted from an intermediate layer of the network.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
%matplotlib inline
```

1.1 Get a pre-trained model

We choose the model described in paper SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size.

[3]:
```
import torchvision.models as models
model = models.squeezenet1_0(pretrained=True)
model
```

[3]:
```
SqueezeNet(
  (features): Sequential(
    (0): Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (3): Fire(
      (squeeze): Conv2d(96, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (4): Fire(
      (squeeze): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
```

```
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (5): Fire(
      (squeeze): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (6): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (7): Fire(
      (squeeze): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (8): Fire(
      (squeeze): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (9): Fire(
      (squeeze): Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (10): Fire(
      (squeeze): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (11): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
```

```
    (12): Fire(
      (squeeze): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Conv2d(512, 1000, kernel_size=(1, 1), stride=(1, 1))
    (2): ReLU(inplace=True)
    (3): AdaptiveAvgPool2d(output_size=(1, 1))
  )
)
```

The model is stored here:

[4]:
```
import os
path = os.path.join(os.environ.get('USERPROFILE', os.environ.get('HOME', '.')),
                    ".cache", "torch", "checkpoints")
if os.path.exists(path):
    res = os.listdir(path)
else:
    res = ['not found', path]
res
```

[4]:
```
['squeezenet1_0-a815701f.pth', 'squeezenet1_1-f364aa15.pth']
```

pytorch's design relies on two methods *forward* and *backward* which implement the propagation and back-propagation of the gradient, the structure is not known and could even be dyanmic. That's why it is difficult to define a number of layers.

[5]:
```
len(model.features), len(model.classifier)
```

[5]:
```
(13, 4)
```

1.2 Images

We collect images from pixabay.

1.2.1 Raw images

[6]:
```
from pyquickhelper.filehelper import unzip_files
if not os.path.exists('simages/category'):
    os.makedirs('simages/category')
files = unzip_files("data/dog-cat-pixabay.zip", where_to="simages/category")
len(files), files[0]
```

[6]:
```
(31, 'simages/category\\cat-1151519__480.jpg')
```

[7]:
```
from mlinsights.plotting import plot_gallery_images
plot_gallery_images(files[:2]);
```

[8]:
```
from torchvision import datasets, transforms

trans = transforms.Compose([transforms.Resize((224, 224)), # essayer avec 224 seulement
                            transforms.CenterCrop(224),
                            transforms.ToTensor()])
imgs = datasets.ImageFolder("simages", trans)
imgs
```

[8]:
```
Dataset ImageFolder
    Number of datapoints: 31
    Root location: simages
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=PIL.Image.BILINEAR)
               CenterCrop(size=(224, 224))
               ToTensor()
           )
```

[9]:
```
from torch.utils.data import DataLoader
dataloader = DataLoader(imgs, batch_size=1, shuffle=False, num_workers=1)
dataloader
```

[9]:
```
<torch.utils.data.dataloader.DataLoader at 0x2c6e4120cc0>
```

[10]:
```
img_seq = iter(dataloader)
img, cl = next(img_seq)
```

[11]:
```
type(img), type(cl)
```

[11]:
```
(torch.Tensor, torch.Tensor)
```

[12]:
```
array = img.numpy().transpose((2, 3, 1, 0))
array.shape
```

[12]:
```
(224, 224, 3, 1)
```

[13]:
```
import matplotlib.pyplot as plt
plt.imshow(array[:,:,:,0])
plt.axis('off');
```

[14]:
```
img, cl = next(img_seq)
array = img.numpy().transpose((2, 3, 1, 0))
plt.imshow(array[:,:,:,0])
plt.axis('off');
```

torch implements optimized function to load and process images.

[15]:
```
trans = transforms.Compose([transforms.Resize((224, 224)), # essayer avec 224 seulement
                            transforms.RandomRotation((-10, 10), expand=True),
```

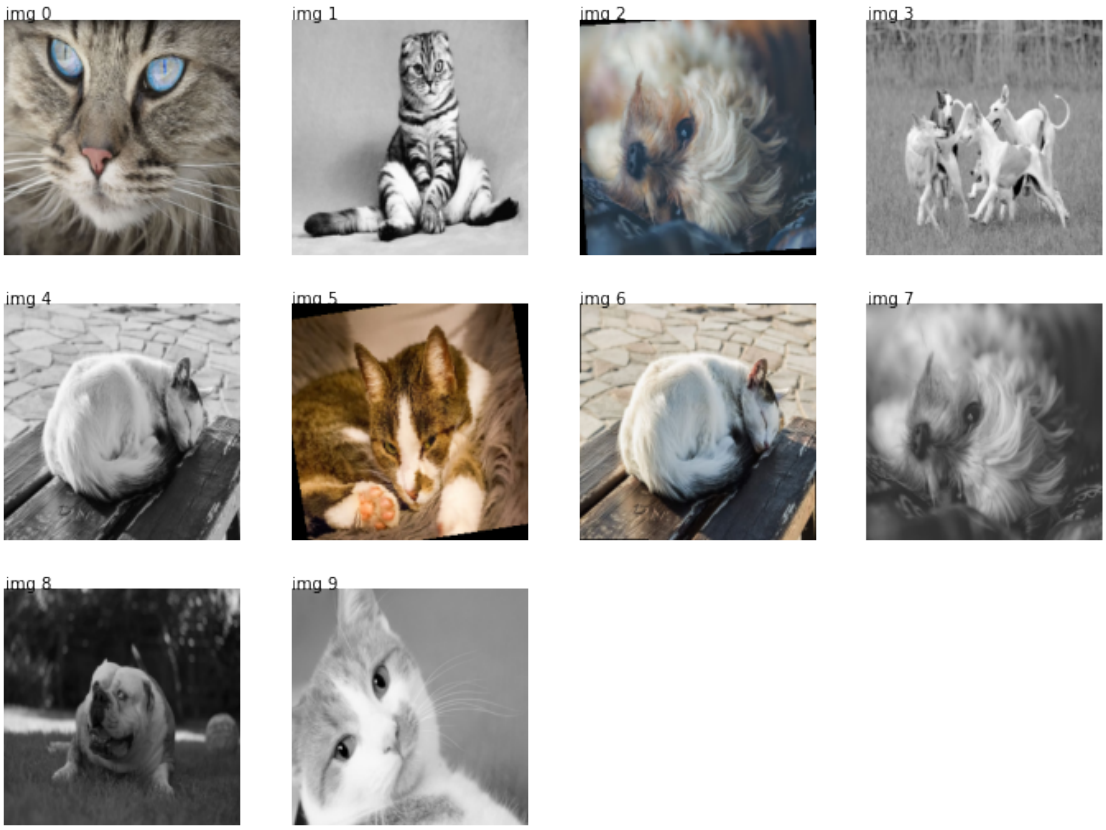
```
                            transforms.CenterCrop(224),
                            transforms.ToTensor(),
                           ])
imgs = datasets.ImageFolder("simages", trans)
dataloader = DataLoader(imgs, batch_size=1, shuffle=True, num_workers=1)
img_seq = iter(dataloader)
imgs = list(img[0] for i, img in zip(range(2), img_seq))
```

[16]:
```
plot_gallery_images([img.numpy().transpose((2, 3, 1, 0))[:,:,:,0] for img in imgs]);
```

We can multiply the data by implementing a custom sampler or just concatenate loaders.

[17]:
```
from torch.utils.data import ConcatDataset
trans1 = transforms.Compose([transforms.Resize((224, 224)), # essayer avec 224␣
 ↪seulement
                             transforms.RandomRotation((-10, 10), expand=True),
                             transforms.CenterCrop(224),
                             transforms.ToTensor()])
trans2 = transforms.Compose([transforms.Resize((224, 224)), # essayer avec 224␣
 ↪seulement
                             transforms.Grayscale(num_output_channels=3),
                             transforms.CenterCrop(224),
                             transforms.ToTensor()])
imgs1 = datasets.ImageFolder("simages", trans1)
imgs2 = datasets.ImageFolder("simages", trans2)
dataloader = DataLoader(ConcatDataset([imgs1, imgs2]), batch_size=1,
                        shuffle=True, num_workers=1)
img_seq = iter(dataloader)
imgs = list(img[0] for i, img in zip(range(10), img_seq))
```

[18]:
```
plot_gallery_images([img.numpy().transpose((2, 3, 1, 0))[:,:,:,0] for img in imgs]);
```



Which leaves 52 images to process out of 61 = 31*2 (the folder contains 31 images).

[19]:
```
len(list(img_seq))
```

[19]: 52

1.3 Search among images

We use the class `SearchEnginePredictionImages`.

1.3.1 The idea of the search engine

The deep network is able to classify images coming from a competition called ImageNet which was trained to classify different images. But still, the network has 88 layers which slightly transform the images into classification results. We assume the last layers contains information which allows the network to classify into objects: it is less related to the images than the content of it. In particular, we would like that an image with a daark background does not necessarily return images with a dark background.
We reshape an image into *(224x224)* which is the size the network ingests. We propagate the inputs until the layer just before the last one. Its output will be considered as the *featurized image*. We do that for a specific set of images called the *neighbors*. When a new image comes up, we apply the same process and find the closest images among the set of neighbors.

[20]:
```
import torchvision.models as models
model = models.squeezenet1_0(pretrained=True)
```

The model outputs the probability for each class.

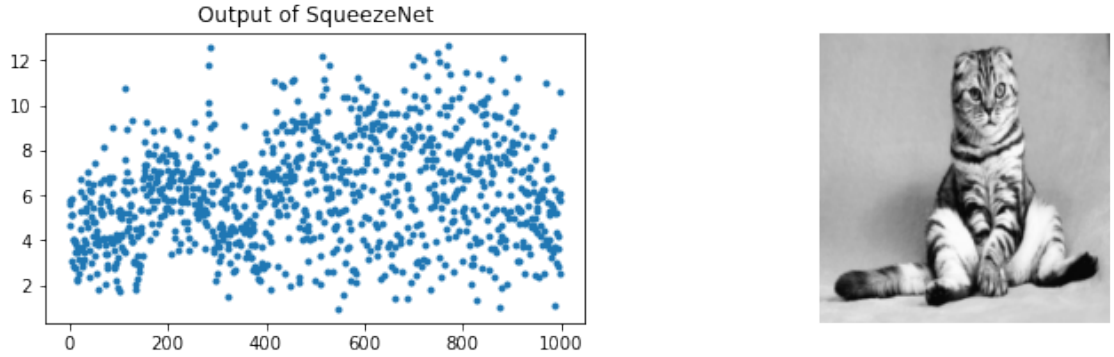
[21]:
```
res = model.forward(imgs[1])
res.shape
```

[21]:
```
torch.Size([1, 1000])
```

[22]:
```
res.detach().numpy().ravel()[:10]
```

[22]:
```
array([5.7371173, 5.61982  , 4.685445 , 5.816555 , 5.151505 , 5.1619806,
       3.1080377, 4.0115213, 4.023687 , 2.8594074], dtype=float32)
```

[23]:
```
fig, ax = plt.subplots(1, 2, figsize=(12,3))
ax[0].plot(res.detach().numpy().ravel(), '.')
ax[0].set_title("Output of SqueezeNet")
ax[1].imshow(imgs[1].numpy().transpose((2, 3, 1, 0))[:,:,:,0])
ax[1].axis('off');
```



We have features for one image. We build the neighbors, the output for each image in the training datasets.

[24]:
```
trans = transforms.Compose([transforms.Resize((224, 224)),
                            transforms.CenterCrop(224),
                            transforms.ToTensor()])
imgs = datasets.ImageFolder("simages", trans)
dataloader = DataLoader(imgs, batch_size=1, shuffle=False, num_workers=1)
img_seq = iter(dataloader)
imgs = list(img[0] for img in img_seq)
```

[25]:
```
all_outputs = [model.forward(img).detach().numpy().ravel() for img in imgs]
```

[26]:
```
from sklearn.neighbors import NearestNeighbors
knn = NearestNeighbors()
knn.fit(all_outputs)
```

[26]:
```
NearestNeighbors()
```

We extract the neighbors for a new image.

[27]:
```
one_output = model.forward(imgs[5]).detach().numpy().ravel()
```

[28]:
```
score, index = knn.kneighbors([one_output])
score, index
```

[28]:
```
(array([[24.470465, 59.278355, 69.84957 , 71.872154, 77.75205 ]],
       dtype=float32),
 array([[ 5,  1,  0,  9, 28]], dtype=int64))
```

We need to retrieve images for indexes stored in *index*.

[29]:
```
import os
names = os.listdir("simages/category")
names = [os.path.join("simages/category", n) for n in names]
disp = [names[5]] + [names[i] for i in index.ravel()]
disp
```

[29]:
```
['simages/category\\cat-2603300__480.jpg',
 'simages/category\\cat-2603300__480.jpg',
 'simages/category\\cat-1192026__480.jpg',
 'simages/category\\cat-1151519__480.jpg',
 'simages/category\\cat-2922832__480.jpg',
 'simages/category\\shotlanskogo-2934720__480.jpg']
```

We check the first one is exactly the same as the searched image.

[30]:
```
plot_gallery_images(disp);
```

It is possible to access intermediate layers output however it means rewriting the method forward to capture it: Accessing intermediate layers of a pretrained network forward?.

1.4 Going further

The original neural network has not been changed and was chosen to be small (88 layers). Other options are available for better performances. The imported model can be also be trained on a classification problem if there is such information to leverage. Even if the model was trained on millions of images, a couple of

thousands are enough to train the last layers. The model can also be trained as long as there exists a way to compute a gradient. We could imagine to label the result of this search engine and train the model on pairs of images ranked in the other.
We can use the pairwise transform (example of code: ranking.py). For every pair (X_i, X_j), we tell if the search engine should have $X_i \prec X_j$ ($Y_{ij} = 1$) or the order order ($Y_{ij} = 0$). X_i is the features produced by the neural network : $X_i = f(\Omega, img_i)$. We train a classifier on the database:

$$(f(\Omega, img_i) - f(\Omega, img_j), Y_{ij})_{ij}$$

A training algorithm based on a gradient will have to propagate the gradient : $\frac{\partial f}{\partial \Omega}(img_i) - \frac{\partial f}{\partial \Omega}(img_j)$.

[31]: