BigData: Architectural Categorization

December 13, 2021

1 DataScraper

[ ]:
```
!pip install selenium
!pip install webdriver_manager
!pip install BeautifulSoup4
```

[ ]:
```
import os
import re
from selenium import webdriver
import time
from webdriver_manager.chrome import ChromeDriverManager
from bs4 import BeautifulSoup
import urllib.request
```

Chrome Driver

The raw html data of a google search has a very limited amount of images.

That is why we use a chrome driver (from selenium library) to open a chrome browser tab and make a search on google images. Then we tell the driver to scroll down a few times and lastly it gives us the raw html data that is loaded.

We have tried programming around the limit that google images sets for scrapers, but we weren't succesfull and it made it a lot slower.

Also we run this driver on a larger screen so more images would be loaded per scroll.

[ ]:
```
def getHtml(query,scrolls = 10):
    driver = webdriver.Chrome(ChromeDriverManager().install())

    search_url="https://www.google.com/search?
 ↪q={q}&tbm=isch&tbs=sur%3Afc&hl=en&ved=0CAIQpwVqFwoTCKCa1c6s4-oCFQAAAAAdAAAAABAC&biw=1251&bih=

    driver.get(search_url.format(q=query))

    #Scroll to the end of the page
    scrollsDone = 0
    clicked = False
    while scrollsDone<scrolls:
```

```
        print(scrolls)
        try:
            driver.find_element_by_xpath("//span[@jsaction= 'h5M12e']").click()
            clicked=True
            time.sleep(1)
        except:
            print('click failed')
        try:
            driver.find_element_by_xpath("//input[@jsaction= 'Pmjnye']").click()
            clicked=True
            time.sleep(1)
        except:
            print('click failed')
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        time.sleep(2)#sleep_between_interactions
        scrollsDone +=1


    #Locate the images to be scraped from the current page
    html = driver.page_source
    driver.quit
    return html
```

BeautifulSoup

We use BeautifulSoup to read and search the raw html for the right images.

[ ]:
```
def getImages(html):
    soup = BeautifulSoup(html,"html.parser")
    return soup.find_all("img" ,attrs={"src":True})
```

Saving the images

This is a loop that loops through the images data and downloads the image via the url in the src atribute.

[ ]:
```
def saveImages(images,location,name = "",count=-1):
    number = 0
    #first image is the google icon
    for image in images[1:]:
        image_src=image["src"]

        urllib.request.urlretrieve(image_src, location+ name + str(number)+ ".
↪png")
        number += 1
        count -=1
        if count == 0:
            return
```

getData is used to to save n images for 1 query in datafolder with name as folder and name for

every image.

```
[ ]: def getData(query,n,name,datafolder="./Data"):
         images = getImages(getHtml(query,scrolls=round(n/80)+1))
         if not os.path.isdir(datafolder+"/"+name): os.mkdir(datafolder+"/"+name)
         saveImages(images,location=datafolder+"/"+name+ "/", name=name,count=n)
```

This function n amount of images for every query. 1000 just means max, google images controlles the amount of images available.

```
[ ]: def getGoogleImages(querys,n=1000,datafolder="./Data"):
         if not os.path.isdir(datafolder): os.mkdir(datafolder)
         for q in querys:
             getData(query=q,n=n,name=re.sub("␣
      ↪architecture","",q),datafolder=datafolder)
```

The images saved this way are small (100x100 px). This could be an issue if very small details are needed for categorization. But the images are detailed enough to categorize and most categorization models use images from 128x128, not so far of our images. Also this makes scraping the data fast for the amount of pictures, getting higher resolution would require a much longer downloading time. We agreed this was a fair trade-off.

```
[ ]: #"art deco","baroque","gothic","roman","tudor","ancient egyptian" ,␣
      ↪"moorisch","rococo","indoislamic","federal","expressionist","modern","cunstructivist",
     querys = ["art deco","baroque","gothic","roman","tudor","ancient egyptian" ,␣
      ↪"moorisch","rococo","indoislamic","federal","expressionist","modern","cunstructivist","brutal
     addArchitecture = lambda q: q + " architecture"
     getGoogleImages(list(map(addArchitecture,querys)),n=20,datafolder="./static/
      ↪data")
```

2 Model training

Mounten van de google drive, het importeren van fastai en het uitpakken van de foto's gebeurd in de eerste stap.

[4]:
```
!pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

```
     || 720 kB 12.8 MB/s
     || 46 kB 3.6 MB/s
     || 189 kB 45.0 MB/s
     || 1.2 MB 49.2 MB/s
     || 56 kB 4.5 MB/s
     || 51 kB 314 kB/s
Mounted at /content/gdrive
```

[5]:
```
from fastbook import *
from fastai.vision.widgets import *
```

[3]:
```
!unzip data.zip
```

Het tonen van 1 foto om te zien of het pad goed is en de data goed geladen is.

[4]:
```
from PIL import Image
path = '/content/data/'
im = Image.open(path + 'art deco/art deco0.png')
im.to_thumb(128,128)
```

[4]:

[5]:
```
from fastai.vision.all import *
filenames = get_image_files(path)
filenames
```

[5]:
```
(#9053) [Path('/content/data/tudor/tudor479.png'),Path('/content/data/tudor/tudo
r653.png'),Path('/content/data/tudor/tudor114.png'),Path('/content/data/tudor/tu
dor665.png'),Path('/content/data/tudor/tudor572.png'),Path('/content/data/tudor/
tudor184.png'),Path('/content/data/tudor/tudor285.png'),Path('/content/data/tudo
r/tudor134.png'),Path('/content/data/tudor/tudor499.png'),Path('/content/data/tu
```

```
dor/tudor581.png')...]
```

Nakijken of er geen corrupte files tussen staan

[6]:
```
failed = verify_images(filenames)
failed
```

[6]:
```
(#0) []
```

dataloaders

Quick summary of the doc:

[6]:
```
??DataLoaders
```

fastai-data-block-api

[8]:
```
architecture = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(128))
```

[9]:
```
doc(DataBlock)
```

```
<IPython.core.display.HTML object>
```

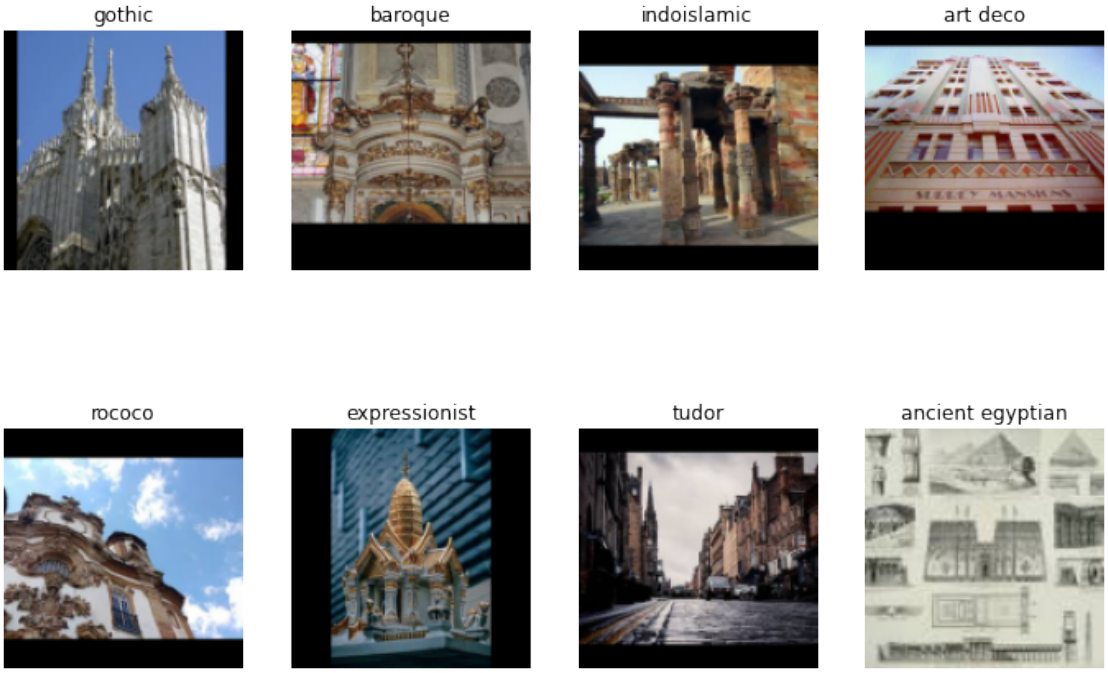
[11]:
```
dls = architecture.dataloaders(path)
```

[12]:
```
dls.train.show_batch(max_n=4, nrows=1)
dls.valid.show_batch(max_n=4, nrows=1)
```

Tonen van 8 foto's 4 per row die ge resized zijn

[13]:
```
architecture = architecture.new(item_tfms=Resize(128, ResizeMethod.Squish))
dls = architecture.dataloaders(path)
dls.train.show_batch(max_n=4, nrows=1)
dls.valid.show_batch(max_n=4, nrows=1)
```

[14]:
```
architecture = architecture.new(item_tfms=Resize(128, ResizeMethod.Pad,␣
 ↪pad_mode='zeros'))
dls = architecture.dataloaders(path)
dls.train.show_batch(max_n=4, nrows=1)
dls.valid.show_batch(max_n=4, nrows=1)
```



[15]:
```
architecture = architecture.new(item_tfms=RandomResizedCrop(128, min_scale=0.3))
dls = architecture.dataloaders(path)
dls.train.show_batch(max_n=4, nrows=1, unique=True)
# the unique=True piece here is just to force our sanity check to give us the
 ↪same image over and over again,
# ofcourse each time with the random transform applied to it
```

Het trainen van het model kan eindelijk gebeuren. We hebben gekozen voor maar 3 epochs aangezien vanaf 4 epochs de data biased werd.

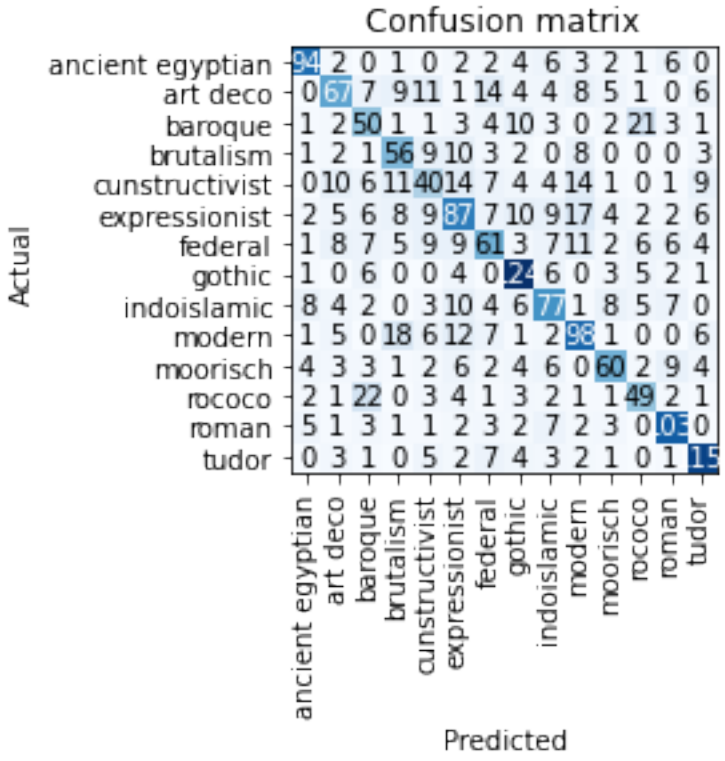
[18]:
```
our_out_of_the_box_model = cnn_learner(dls, resnet50, metrics=error_rate)
our_out_of_the_box_model.fine_tune(3)
```

```
<IPython.core.display.HTML object>
```

<IPython.core.display.HTML object>

[19]:
```
interp = ClassificationInterpretation.from_learner(our_out_of_the_box_model)
interp.plot_confusion_matrix()
```

<IPython.core.display.HTML object>



google-teachable-machine-verwarringsmatrix

[20]:
```
#interp.plot_top_losses(2, nrows=1)

def plot_top_losses_fix(interp, k, largest=True, **kwargs):
        losses,idx = interp.top_losses(k, largest)
        if not isinstance(interp.inputs, tuple): interp.inputs = (interp.inputs,)
        if isinstance(interp.inputs[0], Tensor): inps = tuple(o[idx] for o in␣
 ↪interp.inputs)
        else: inps = interp.dl.create_batch(interp.dl.before_batch([tuple(o[i]␣
 ↪for o in interp.inputs) for i in idx]))
        b = inps + tuple(o[idx] for o in (interp.targs if is_listy(interp.targs)␣
 ↪else (interp.targs,)))
        x,y,its = interp.dl._pre_show_batch(b, max_n=k)
```

```
        b_out = inps + tuple(o[idx] for o in (interp.decoded if is_listy(interp.
↪decoded) else (interp.decoded,)))
        x1,y1,outs = interp.dl._pre_show_batch(b_out, max_n=k)
        if its is not None:
            #plot_top_losses(x, y, its, outs.itemgot(slice(len(inps), None)),␣
↪L(self.preds).itemgot(idx), losses,  **kwargs)
            plot_top_losses(x, y, its, outs.itemgot(slice(len(inps), None)),␣
↪interp.preds[idx], losses,  **kwargs)
        #TODO: figure out if this is needed
        #its None means that a batch knows how to show itself as a whole, so we␣
↪pass x, x1
        #else: show_results(x, x1, its, ctxs=ctxs, max_n=max_n, **kwargs)
```

[21]:
```
plot_top_losses_fix(interp, 10, nrows=2)
```

[22]:
```
# saving our model, by default in a folder called 'models'.
our_out_of_the_box_model.save('good_model')
#creating an serialized pickle object of our model, the export.pkl file
our_out_of_the_box_model.export()
```

[22]:
```
ls
```

```
data/  data.zip  export.pkl  gdrive/
models/  sample_data/
```

loading-a-model-inference

[24]:
```
our_out_of_the_box_model_inference = load_learner('export.pkl')
# let's test our model on an image
our_out_of_the_box_model_inference.predict('/content/data/modern/modern101.png')
# this will return the predicted category, the index of this predicted category,␣
 ↪and the probabilities of each category
```

```
<IPython.core.display.HTML object>
```

[24]:
```
('expressionist',
 TensorBase(5),
 TensorBase([4.7753e-03, 2.5152e-02, 5.1793e-04, 4.0738e-02, 4.2338e-03,
6.1465e-01, 4.5408e-02, 2.2867e-03, 9.1173e-03, 2.5046e-01, 6.2986e-04,
5.1209e-04, 1.3096e-03, 2.1098e-04]))
```

[ ]:
```
our_out_of_the_box_model_inference.dls.vocab
```

---

[1]:
```
!pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

```
     || 720 kB 5.1 MB/s
     || 189 kB 45.7 MB/s
     || 46 kB 3.9 MB/s
     || 1.2 MB 36.0 MB/s
     || 56 kB 4.4 MB/s
     || 51 kB 286 kB/s
Mounted at /content/gdrive
```

[2]:
```
from fastbook import *
```

[3]:
```
!unzip data.zip
```

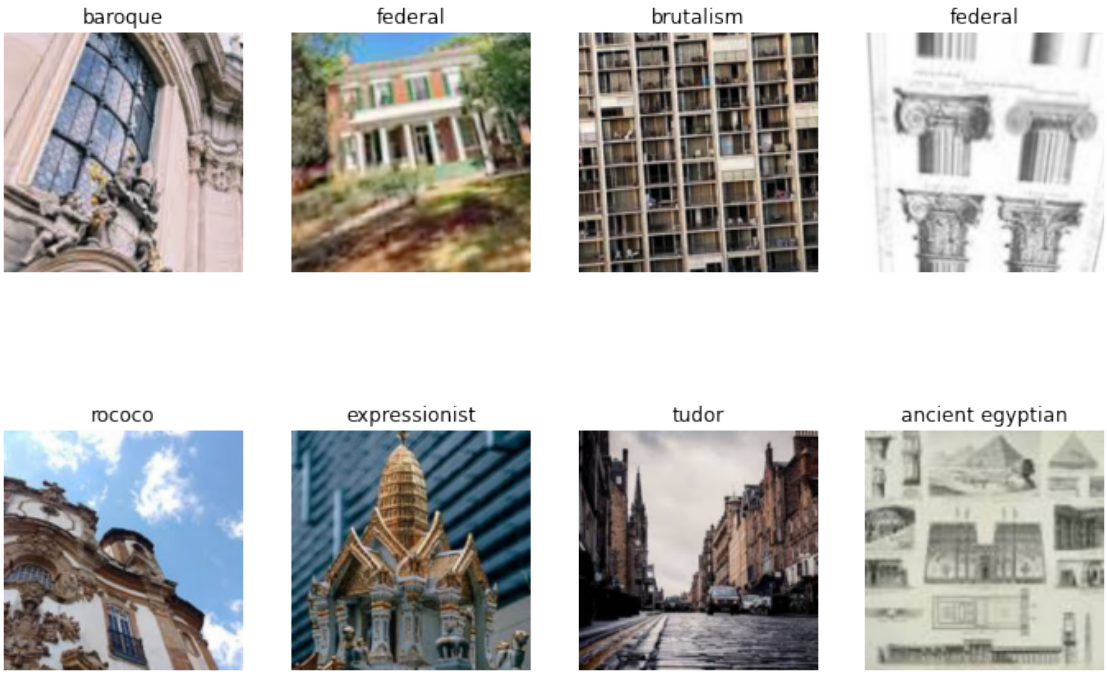
[10]:
```
Gebouwen = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    # the line of code below is just using Regular Expressions to link a file to␣
 ↪a label
    # the label, the breed of the pet, is in the filename, that's why we need a␣
 ↪re to extract it as the label
    get_y=parent_label,
      # now let's also add some awesome augmentation (presizing) into the mix as␣
 ↪well
    item_tfms=Resize(460),
    batch_tfms=aug_transforms(size=224, min_scale=0.75))
```

[11]:
```
path = '/content/data/'
dls = Gebouwen.dataloaders(path)
```

```
/usr/local/lib/python3.7/dist-packages/torch/_tensor.py:1051: UserWarning:
torch.solve is deprecated in favor of torch.linalg.solveand will be removed in a
future PyTorch release.
torch.linalg.solve has its arguments reversed and does not return the LU
```

```
factorization.
To get the LU factorization see torch.lu, which can be used with torch.lu_solve
or torch.lu_unpack.
X = torch.solve(B, A).solution
should be replaced with
X = torch.linalg.solve(A, B) (Triggered internally at
../aten/src/ATen/native/BatchLinearAlgebra.cpp:766.)
  ret = func(*args, **kwargs)
```

[12]:
```
dls.train.show_batch(max_n=4, nrows=1)
dls.valid.show_batch(max_n=4, nrows=1)
```



[13]:
```
alex_model = cnn_learner(dls, alexnet, metrics=error_rate)
alex_model.fine_tune(1)
```

```
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to
/root/.cache/torch/hub/checkpoints/alexnet-owt-7be5be79.pth

  0%|          | 0.00/233M [00:00<?, ?B/s]

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[14]:
```
vgg16_model = cnn_learner(dls, vgg16_bn, metrics=error_rate)
vgg16_model.fine_tune(1)
```

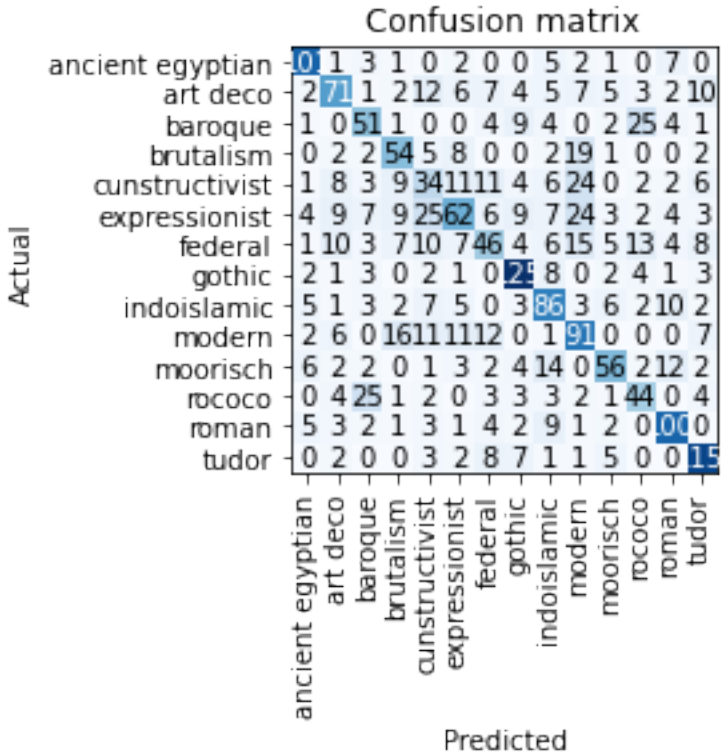
```
Downloading: "https://download.pytorch.org/models/vgg16_bn-6c64b313.pth" to
/root/.cache/torch/hub/checkpoints/vgg16_bn-6c64b313.pth

  0%|          | 0.00/528M [00:00<?, ?B/s]

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[15]:
```
resnet50_model = cnn_learner(dls, resnet50, metrics=error_rate)
resnet50_model.fine_tune(1)
```

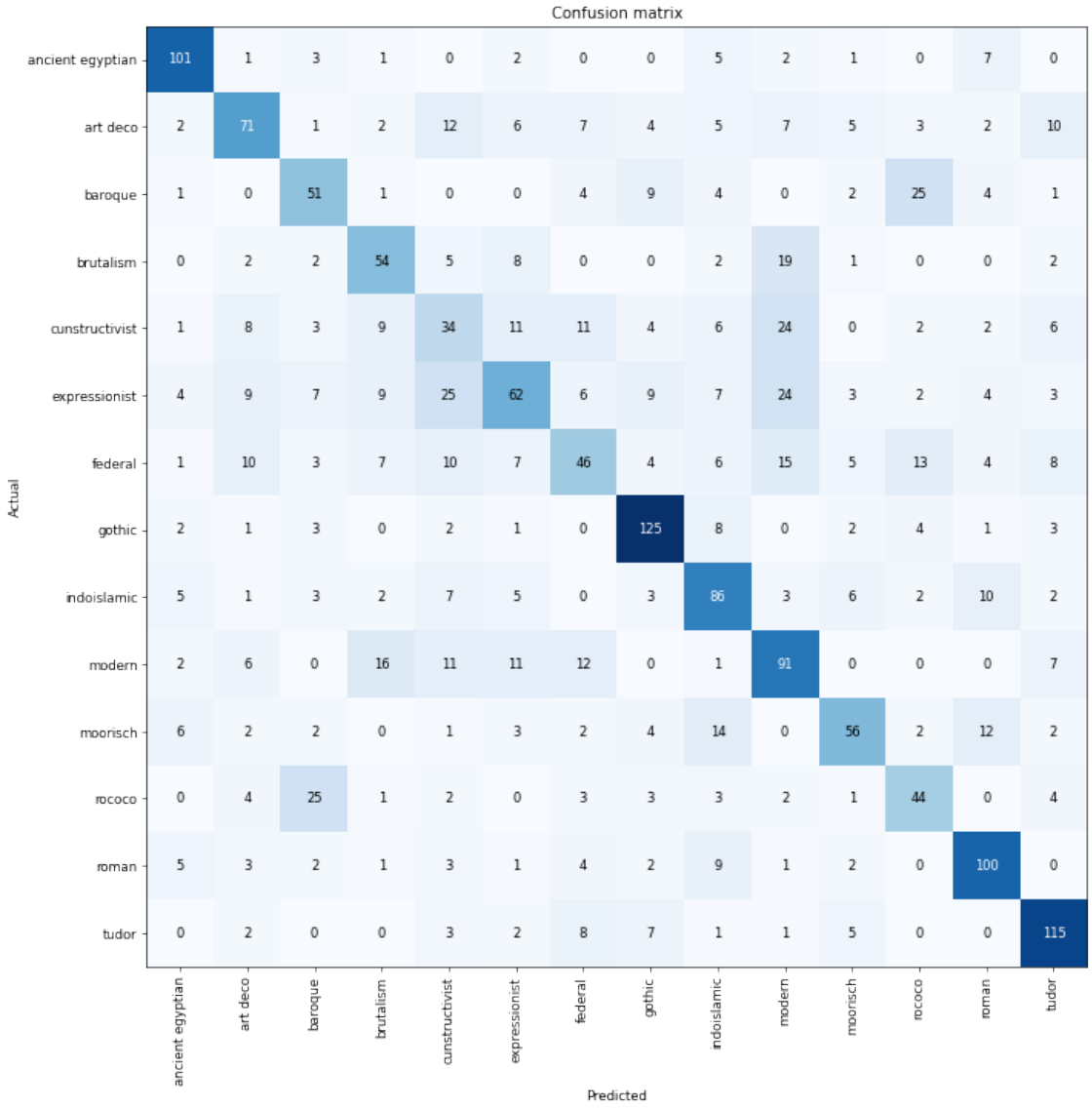
```
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to
/root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth

  0%|          | 0.00/97.8M [00:00<?, ?B/s]

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[16]:
```
interp = ClassificationInterpretation.from_learner(resnet50_model)
interp.plot_confusion_matrix()
```

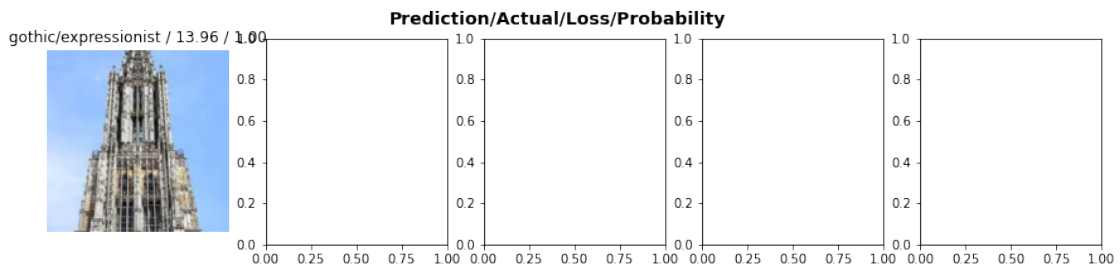
```
<IPython.core.display.HTML object>
```



[17]:
```
interp.plot_confusion_matrix(figsize=(12,12), dpi=60)
```



[18]:
```
interp.plot_top_losses(5, nrows=1)
```



[19]:
```
interp.most_confused(min_val=5)
```

[19]:
```
[('baroque', 'rococo', 25),
 ('expressionist', 'cunstructivist', 25),
 ('rococo', 'baroque', 25),
 ('cunstructivist', 'modern', 24),
 ('expressionist', 'modern', 24),
 ('brutalism', 'modern', 19),
 ('modern', 'brutalism', 16),
 ('federal', 'modern', 15),
 ('moorisch', 'indoislamic', 14),
 ('federal', 'rococo', 13),
 ('art deco', 'cunstructivist', 12),
 ('modern', 'federal', 12),
 ('moorisch', 'roman', 12),
 ('cunstructivist', 'expressionist', 11),
 ('cunstructivist', 'federal', 11),
 ('modern', 'cunstructivist', 11),
 ('modern', 'expressionist', 11),
 ('art deco', 'tudor', 10),
 ('federal', 'art deco', 10),
 ('federal', 'cunstructivist', 10),
 ('indoislamic', 'roman', 10),
 ('baroque', 'gothic', 9),
 ('cunstructivist', 'brutalism', 9),
 ('expressionist', 'art deco', 9),
 ('expressionist', 'brutalism', 9),
 ('expressionist', 'gothic', 9),
 ('roman', 'indoislamic', 9),
 ('brutalism', 'expressionist', 8),
 ('cunstructivist', 'art deco', 8),
 ('federal', 'tudor', 8),
 ('gothic', 'indoislamic', 8),
 ('tudor', 'federal', 8),
 ('ancient egyptian', 'roman', 7),
 ('art deco', 'federal', 7),
 ('art deco', 'modern', 7),
 ('expressionist', 'baroque', 7),
 ('expressionist', 'indoislamic', 7),
 ('federal', 'brutalism', 7),
 ('federal', 'expressionist', 7),
 ('indoislamic', 'cunstructivist', 7),
 ('modern', 'tudor', 7),
 ('tudor', 'gothic', 7),
```

```
 ('art deco', 'expressionist', 6),
 ('cunstructivist', 'indoislamic', 6),
 ('cunstructivist', 'tudor', 6),
 ('expressionist', 'federal', 6),
 ('federal', 'indoislamic', 6),
 ('indoislamic', 'moorisch', 6),
 ('modern', 'art deco', 6),
 ('moorisch', 'ancient egyptian', 6),
 ('ancient egyptian', 'indoislamic', 5),
 ('art deco', 'indoislamic', 5),
 ('art deco', 'moorisch', 5),
 ('brutalism', 'cunstructivist', 5),
 ('federal', 'moorisch', 5),
 ('indoislamic', 'ancient egyptian', 5),
 ('indoislamic', 'expressionist', 5),
 ('roman', 'ancient egyptian', 5),
 ('tudor', 'moorisch', 5)]
```

[20]:
```
resnet34_overtrain = cnn_learner(dls, resnet34, metrics=error_rate)
resnet34_overtrain.fine_tune(1, base_lr=0.1)
```

```
Downloading: "https://download.pytorch.org/models/resnet34-b627a593.pth" to
/root/.cache/torch/hub/checkpoints/resnet34-b627a593.pth

  0%|          | 0.00/83.3M [00:00<?, ?B/s]
```

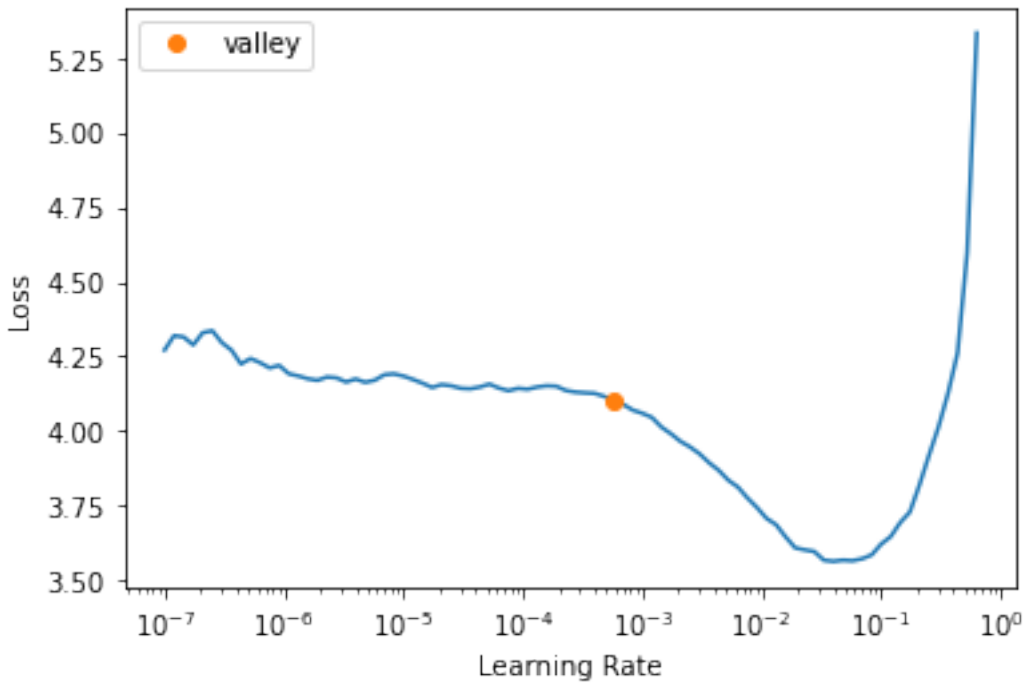
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[21]:
```
resnet34_just_right = cnn_learner(dls, resnet34, metrics=error_rate)
lr_min,lr_steep = resnet34_just_right.lr_find()
```

```
<IPython.core.display.HTML object>
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-182f03c43633> in <module>()
      1 resnet34_just_right = cnn_learner(dls, resnet34, metrics=error_rate)
----> 2 lr_min,lr_steep = resnet34_just_right.lr_find()

ValueError: not enough values to unpack (expected 2, got 1)
```



[22]:
```
print(f"Steepest point: {lr_steep:.2e}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-22-c845ce598eda> in <module>()
----> 1 print(f"Steepest point: {lr_steep:.2e}")

NameError: name 'lr_steep' is not defined
```

[23]:
```
resnet_adv = cnn_learner(dls, resnet34, metrics=error_rate)
resnet_adv.fit_one_cycle(3, 3e-3)
```

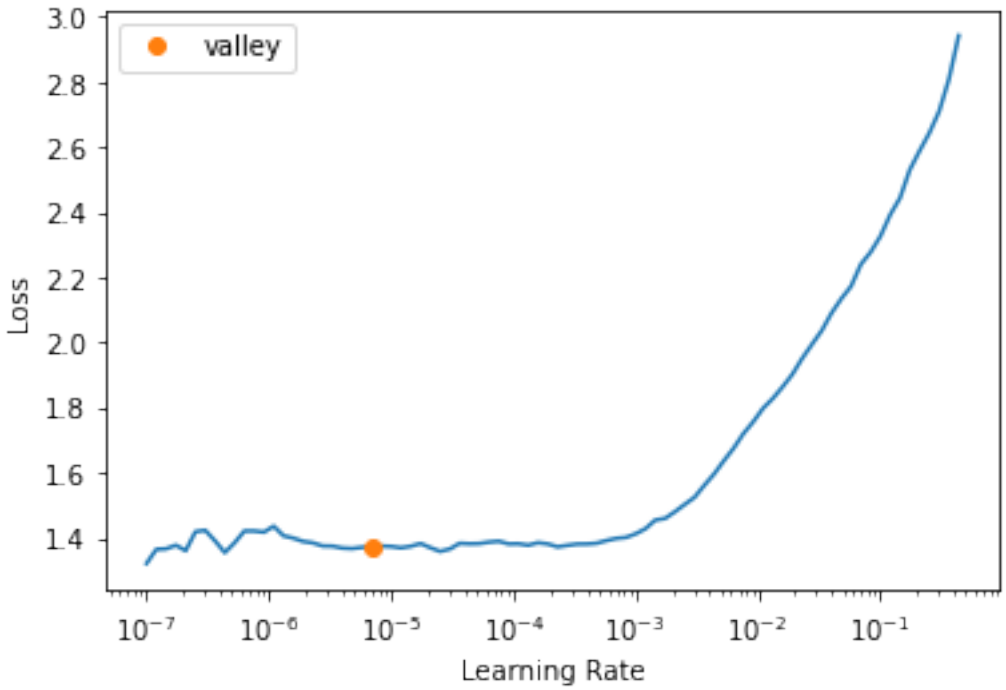
```
<IPython.core.display.HTML object>
```

[24]:
```
resnet_adv.unfreeze()
```

[25]:
```
resnet_adv.lr_find()
```

```
<IPython.core.display.HTML object>
```

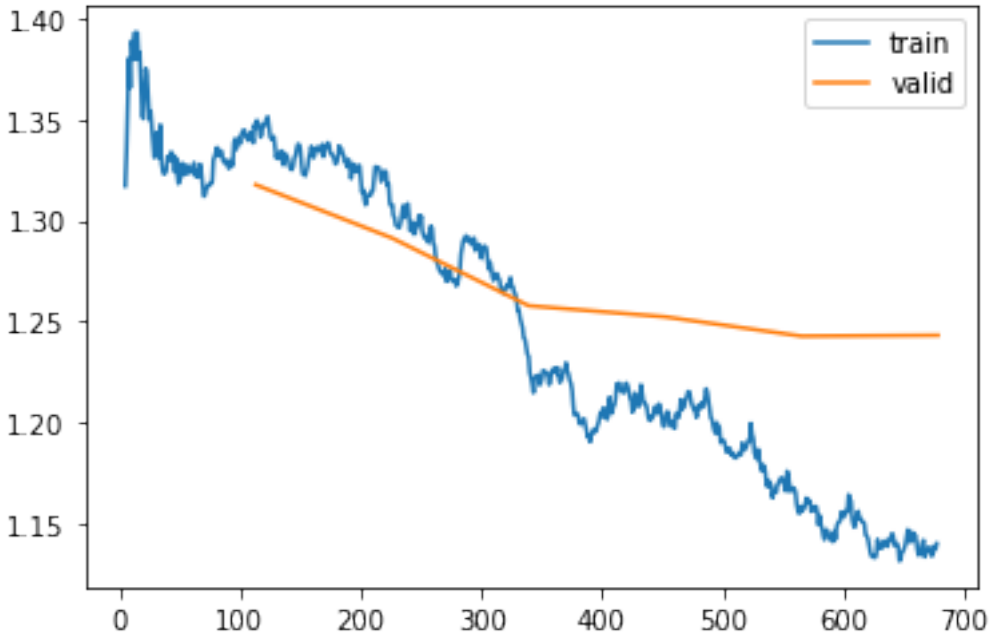
[25]:
```
SuggestedLRs(valley=6.918309736647643e-06)
```



[26]:
```
resnet_adv.fit_one_cycle(6, lr_max=1e-5)
```

```
<IPython.core.display.HTML object>
```

[27]:
```
resnet_adv.recorder.plot_loss()
```

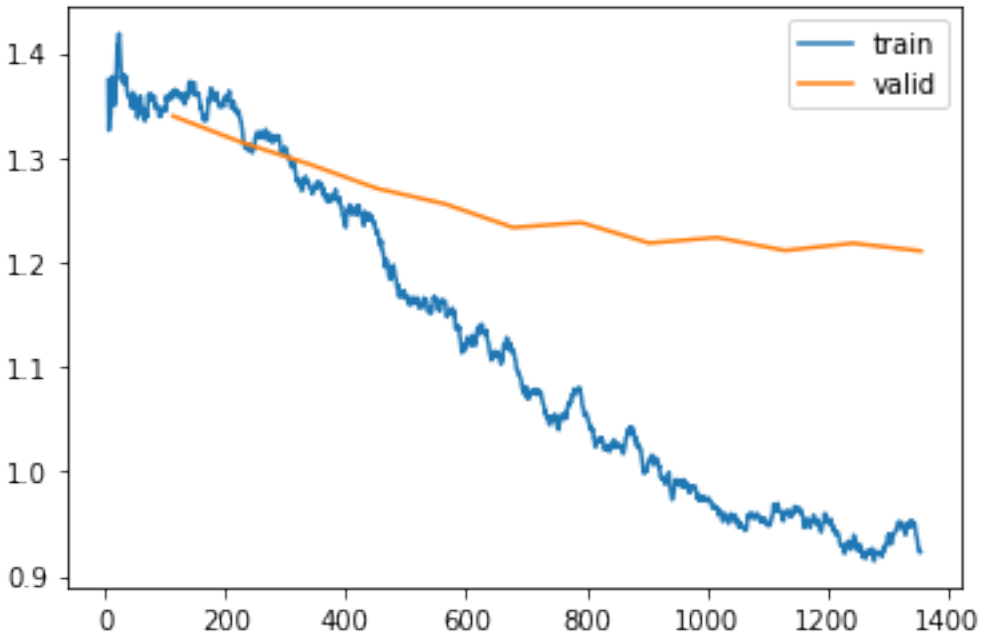


[28]:
```
resnet_really_adv = cnn_learner(dls, resnet34, metrics=error_rate)
resnet_really_adv.fit_one_cycle(3, 3e-3)
resnet_really_adv.unfreeze()
resnet_really_adv.fit_one_cycle(12, lr_max=slice(1e-6,1e-4))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[29]:
```
resnet_really_adv.recorder.plot_loss()
```

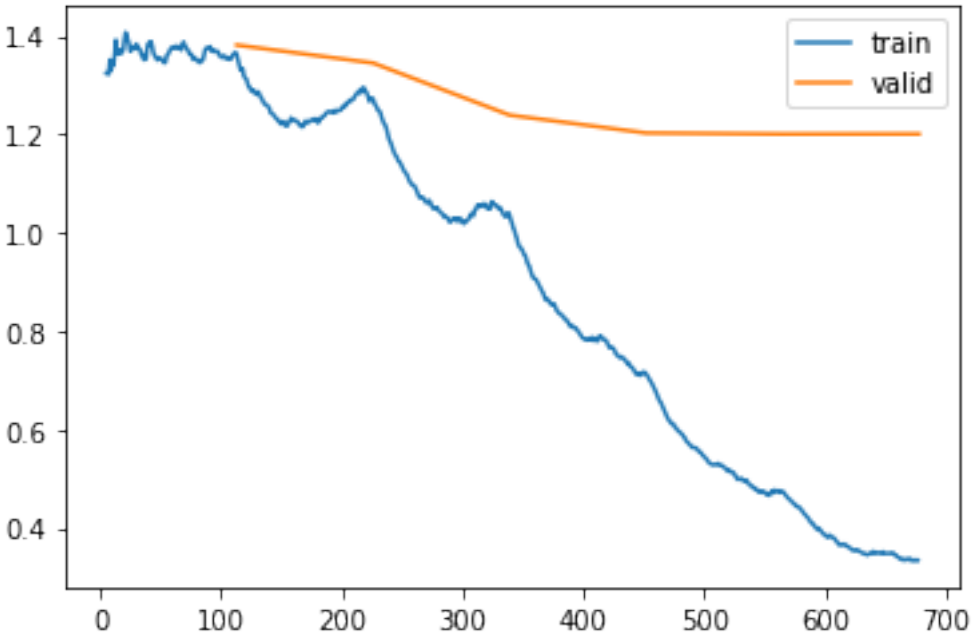


[30]:
```
from fastai.callback.fp16 import *
resnet_adv_tweaked = cnn_learner(dls, resnet50, metrics=error_rate).to_fp16()
resnet_adv_tweaked.fine_tune(6, freeze_epochs=3)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[31]:
```
resnet_adv_tweaked.recorder.plot_loss()
```



[33]:
```
resnet_really_adv.export()
```

3 Flask

Dependencies

[14]:
```
!pip install fastai
!pip install flask
```

Imports

The imports for the flask website. We used os to make directories and re to do some regex on the uploaded files.

There was also a dependency issue with load_learner from fastai.vision.all, tt used PosixPath but that isn't used in windows so we fixed it by replacing PosixPath with the WindowsPath.

[17]:
```
from flask import Flask, render_template, request, redirect, url_for
from werkzeug.utils import secure_filename
import os
import re


import random as rnd
from fastai import *
from fastai.vision.all import *

# fixes a dependancy on PosixPath by load_learner
import pathlib
temp = pathlib.PosixPath
pathlib.PosixPath = pathlib.WindowsPath
```

Global variables

These contain fixed data that is used by the website. The image that is uploaded on the website to be tested by the model is saved in the uploadfolder.

[18]:
```
app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = './static/UploadFolder'

DATAPATH = "./static/data/"
# These are the categories used in the model and the data
CATEGORIES = os.listdir(DATAPATH)
ALLOWED_IMGTYPES = ['png','jpeg','gif','jfif','jpg']
# uses top 20 images from the scrapers
N_MAX = 20

# learn is the model that is loaded from the pkl file. this file contains a␣
 ↪model that predicts 14 different archtectal styles
learn = load_learner('exportBig.pkl')
```

Format functions

These functions are used to format the data that has to be represented in the webpage. Example is a 'card' containing the right category, the image and the predicted category.

Sample is the formatting of the uploaded image to be tested.

[19]:
```
def format_example(imgpath,cat,prediction):
   return {"image":"/data/"+imgpath,
                           "category":cat,
                           "prediction":prediction[0],
                           "correct_class":"example_"+str(cat==prediction[0])}

def format_sample(imgpath,prediction):
   return {"image":"/UploadFolder/"+imgpath,
             "prediction":prediction[0],
             "sample_class":"sample_visable"}
```

Examples function

This function gets a random image from every category and returns a list of all the formatted examples.

[20]:
```
def get_examples():
   examples = []
   for cat in CATEGORIES:
      n = rnd.choice(range(N_MAX))
      imgpath = cat+"/"+cat+str(n)+".png"
      prediction = learn.predict(DATAPATH+imgpath)

      examples.append(format_example(imgpath,cat,prediction))

   return examples
```

Flask routed functions

These functions are routed by Flask and return the same template (home). The difference is that the initial routing is used when no image is uploaded and the second function is used when an image is uploaded.

[21]:
```
@app.route('/')
def home():
   examples = get_examples()
   return render_template('home.html',examples=examples,sample={"sample_class":
 ↪"sample_hidden"})

@app.route("/testImage_post", methods=['post'])
def testImage_post():

   # clears the upload folder, to prevent to much space being used
   for img in os.listdir(app.config['UPLOAD_FOLDER']):
      os.remove(os.path.join(app.config['UPLOAD_FOLDER'],img))
```

```
    # requests file from the posted files, this is suposed to be an image
    image = request.files['file']
    imgpath = secure_filename(image.filename)

    #regex function to see if uploaded file was an image, else return to home␣
↪function
    if re.sub('.*\\.','',imgpath) not in ALLOWED_IMGTYPES:
        return redirect(url_for('home'))

    image.save(os.path.join(app.config['UPLOAD_FOLDER'], imgpath))

    #test the posted image with the model
    prediction = learn.predict(os.path.join(app.config['UPLOAD_FOLDER'], imgpath))

    sample = format_sample(imgpath,prediction)
    examples = get_examples()
    return render_template('home.html',examples=examples,sample=sample)
```

Run the server

[22]:
```
if __name__ == '__main__':
    app.run()
```

```
 * Serving Flask app '__main__' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```