Assignment 2, Question 1

Group [fill in group number]

- **Student 1** : Tommaso Bonomo 1511831
- **Student 2** : Mattia Molon 1511866

Reading material

- [1] *Artem Babenko, Anton Slesarev, Alexandr Chigorin, Victor Lempitsky, "Neural Codes for Image Retrieval"*, ECCV, 2014. https://arxiv.org/abs/1404.1777 (https://arxiv.org/abs/1404.1777).

NOTE When submitting your notebook, please make sure that the training history of your model is visible in the output. This means that you should **NOT** clean your output cells of the notebook. Make sure that your notebook runs without errors in linear order.

Image Retrieval with Neural Codes

In this task, we are trying the approach proposed in [1], meaning we are using the representations learned by a ConvNet for image retrieval. In particular, we are going to

1. Train and evaluate a ConvNet on an image dataset.
2. Compute the outputs of intermediate layers for a new image dataset, which has not been used during training. These values serve as a representation, so-called *neural codes* for the new images.
3. Use the neural codes for image retrieval, by comparing the Euclidean distances between the codes of a query image and the remaining images.
4. Evaluate results both qualitatively and in terms of *mean average precision*.

In [0]:
```
%matplotlib inline

import os
import multiprocessing
import shutil
from google.colab import drive

import numpy as np
import pickle

import keras.backend as K
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Flatten, Input, Dense, Conv2D, MaxPooling2D, ReLU, Dropout, Reshape, UpSampling2D,
BatchNormalization
from tensorflow.keras import losses, optimizers
from tensorflow.keras.utils import plot_model

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

import time

num_train_classes = 190
```

```
Using TensorFlow backend.
```

Mount Google Drive

We will save our model there, in the folder deeplearning2020_ass2_task1. **The model is rather big, so please make sure you have about 1 GB of space in your Google Drive.**

In [0]:
```
if not os.path.isdir('drive'):
  drive.mount('drive')
else:
  print('drive already mounted')

base_path = os.path.join('drive', 'My Drive', 'Università', 'Deep Learning', 'Assignment 2', 'deeplearning2020_ass2_task1')
if not os.path.isdir(base_path):
  os.makedirs(base_path)
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at drive
```

Download Tiny Imagenet

Tiny Imagenet is small subset of the original Imagenet dataset (http://www.image-net.org/ (http://www.image-net.org/)), which is one of the most important large scale image classification datasets. Tiny Imagenet has 200 classes, and contains 500 training examples for each class, i.e. 100,000 training examples in total. The images are of dimensions 64x64. **Note: You will need to re-download the data, when your Colab session has been disconnected (i.e. re-evaluate this cell).**

In [0]:
```
# get tiny imagenet
if not os.path.isdir('tiny-imagenet-200'):
  start_time = time.time()
  if not os.path.isfile('tiny-imagenet-200.zip'):
    ! wget "http://cs231n.stanford.edu/tiny-imagenet-200.zip"
  ! unzip -q tiny-imagenet-200.zip -d .
  print("Unzipped.")
  print("Elapsed time: {} seconds.".format(time.time()-start_time))
else:
  print('Found folder tiny-imagenet-200')
```

```
--2020-05-28 17:25:27--  http://cs231n.stanford.edu/tiny-imagenet-200.zip
Resolving cs231n.stanford.edu (cs231n.stanford.edu)... 171.64.68.10
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.68.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 248100043 (237M) [application/zip]
Saving to: ‘tiny-imagenet-200.zip’

tiny-imagenet-200.z 100%[===================>] 236.61M  16.6MB/s    in 18s

2020-05-28 17:25:46 (12.9 MB/s) - ‘tiny-imagenet-200.zip’ saved [248100043/248100043]

Unzipped.
Elapsed time: 34.45748949050903 seconds.
```

Load Tiny Imagenet

We are going to use the official training set of Tiny Imagenet for our purposes, and ignore the official validation and test sets. We will use 190 classes (95,000 images) of the official training set to make new training, validation and test sets, containing 76,000, 9,500, and 9,500 images, respectively. The remaining 10 classes (5,000 images) will never have been seen during training, and will constitute an *out-of-domain* (ood) set. The ood set will be used for image retrieval. Thus, we'll have:

- Train data: 76,000 images, 64x64x3 pixels, classes 0-189
- Validation data: 9,500 images, 64x64x3 pixels, classes 0-189
- Test data: 9,500 images, 64x64x3 pixels, classes 0-189
- Out-of-domain data: 5000 images, 64x64x3 pixels, **classes 190-199**

In [0]:

```
def load_imagenet(num_train_classes):
  def load_class_images(class_string, label):
    """
    Loads all images in folder class_string.

    :param class_string: image folder (e.g. 'n01774750')
    :param label: label to be assigned to these images
    :return class_k_img: (num_files, width, height, 3) numpy array containing
                         images of folder class_string
    :return class_k_labels: numpy array containing labels
    """
    class_k_path = os.path.join('tiny-imagenet-200/train/', class_string, 'images')
    file_list = sorted(os.listdir(class_k_path))

    dtype = np.uint8

    class_k_img = np.zeros((len(file_list), 64, 64, 3), dtype=dtype)
    for l, f in enumerate(file_list):
      file_path = os.path.join('tiny-imagenet-200/train/', class_string, 'images', f)
      img = mpimg.imread(file_path)
      if len(img.shape) == 2:
        class_k_img[l, :, :, :] = np.expand_dims(img, -1).astype(dtype)
      else:
        class_k_img[l, :, :, :] = img.astype(dtype)

    class_k_labels = label * np.ones(len(file_list), dtype=dtype)

    return class_k_img, class_k_labels

  # get the word description for all imagenet 82115 classes
  all_class_dict = {}
  for k, line in enumerate(open('tiny-imagenet-200/words.txt', 'r')):
    n_id, description = line.split('\t')[:2]
    all_class_dict[n_id] = description

  # this will be the description for our 200 classes
  class_dict = {}

  # we enumerate the classes according to their folder names:
  # 'n01443537' -> 0
  # 'n01629819' -> 1
  # ...
  ls_train = sorted(os.listdir('tiny-imagenet-200/train'))
  img = None
  labels = None
  ood_x = None
  ood_y = None

  # the first num_train_classes will make the training, validation, test sets
  for k in range(num_train_classes):
    # the word descritpion of the current class
    class_dict[k] = all_class_dict[ls_train[k]]
    # load images and labels for current class
    class_k_img, class_k_labels = load_class_images(ls_train[k], k)
    # concatenate all samples and labels
    if img is None:
      img = class_k_img
      labels = class_k_labels
    else:
      img = np.concatenate((img, class_k_img), axis=0)
      labels = np.concatenate((labels, class_k_labels))

  # the remaining classes are the out of domain (ood) set
  for k in range(num_train_classes, 200):
    class_dict[k] = all_class_dict[ls_train[k]]
    class_k_img, class_k_labels = load_class_images(ls_train[k], k)
    if ood_x is None:
      ood_x = class_k_img
      ood_y = class_k_labels
    else:
      ood_x = np.concatenate((ood_x, class_k_img), axis=0)
      ood_y = np.concatenate((ood_y, class_k_labels))

  return img, labels, ood_x, ood_y, class_dict

print('Loading data...')
start_time = time.time()
train_x, train_y, ood_x, ood_y, class_dict = load_imagenet(num_train_classes)
print('Data loaded in {} seconds.'.format(time.time() - start_time))

def split_data(x, y, N):
  x_N = x[0:N, ...]
  y_N = y[0:N]
  x_rest = x[N:, ...]
  y_rest = y[N:, ...]
  return x_N, y_N, x_rest, y_rest

# fix random seed
```

```
np.random.seed(42)

# shuffle
N = train_x.shape[0]
rp = np.random.permutation(N)
train_x = train_x[rp, ...]
train_y = train_y[rp]

# train/validation split 80 - 10 - 10
N_val = int(round(N * 0.1))
N_test = int(round(N * 0.1))
val_x, val_y, train_x, train_y = split_data(train_x, train_y, N_val)
test_x, test_y, train_x, train_y = split_data(train_x, train_y, N_test)

# shuffle ood data
N_ood = ood_x.shape[0]
rp = np.random.permutation(N_ood)
ood_x = ood_x[rp, ...]
ood_y = ood_y[rp]

# convert all data into float32
train_x = train_x.astype(np.float32)
train_y = train_y.astype(np.float32)
val_x = val_x.astype(np.float32)
val_y = val_y.astype(np.float32)
test_x = test_x.astype(np.float32)
test_y = test_y.astype(np.float32)
ood_x = ood_x.astype(np.float32)
ood_y = ood_y.astype(np.float32)

# normalize
train_x /= 255.
val_x /= 255.
test_x /= 255.
ood_x /= 255.

print(train_x.shape)
print(val_x.shape)
print(test_x.shape)
print(ood_x.shape)
```

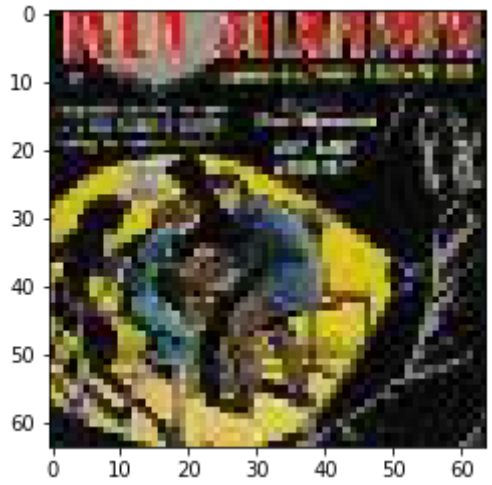
```
Loading data...
Data loaded in 47.61728262901306 seconds.
(76000, 64, 64, 3)
(9500, 64, 64, 3)
(9500, 64, 64, 3)
(5000, 64, 64, 3)
```

Show Some Images

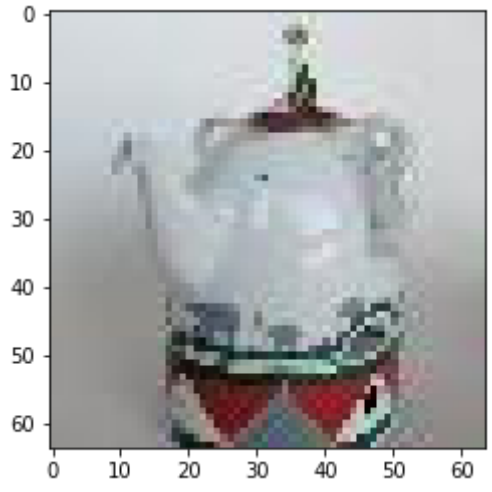
In [0]:
```
def show_random_images(img, labels, K, qualifier):
  for k in range(K):
    idx = np.random.randint(0, img.shape[0])
    print("{} {}: {}".format(qualifier, idx, class_dict[labels[idx]]))
    plt.imshow(img[idx,:,:,:])
    plt.show()

show_random_images(train_x, train_y, 3, 'train')
show_random_images(val_x, val_y, 3, 'validation')
show_random_images(test_x, test_y, 3, 'test')
show_random_images(ood_x, ood_y, 3, 'out of domain')
```

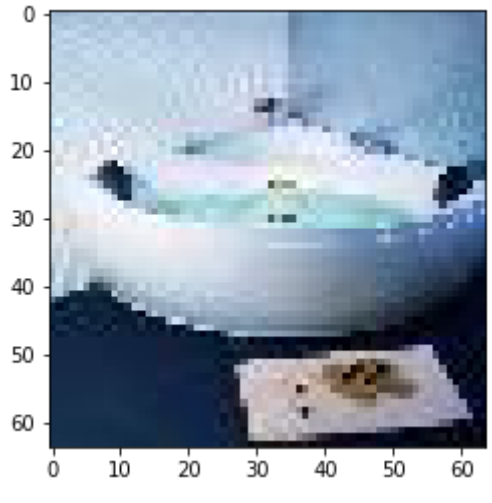
train 30860: comic book



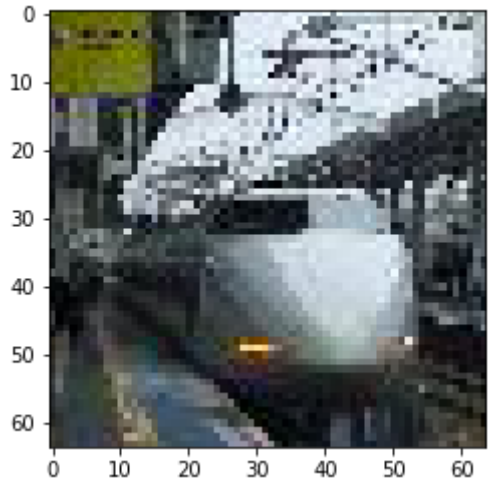
train 21145: teapot



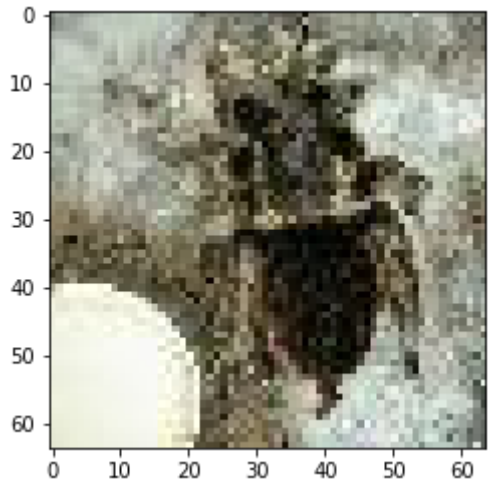
train 37042: bathtub, bathing tub, bath, tub



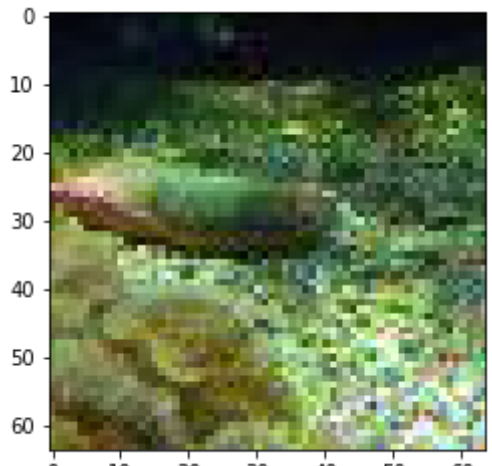
validation 3224: bullet train, bullet



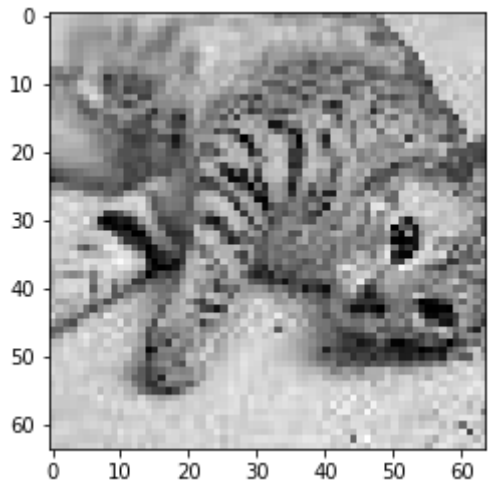
validation 7735: tarantula



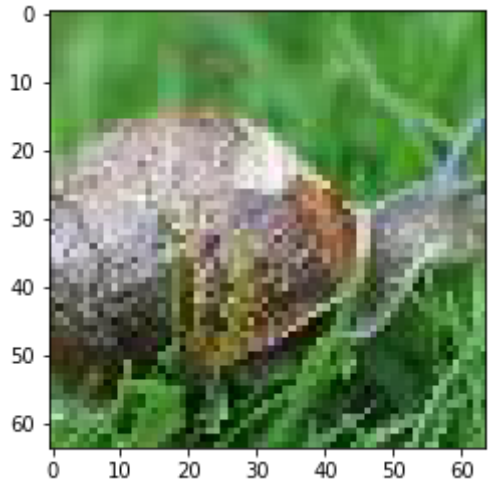
validation 7974: sea cucumber, holothurian



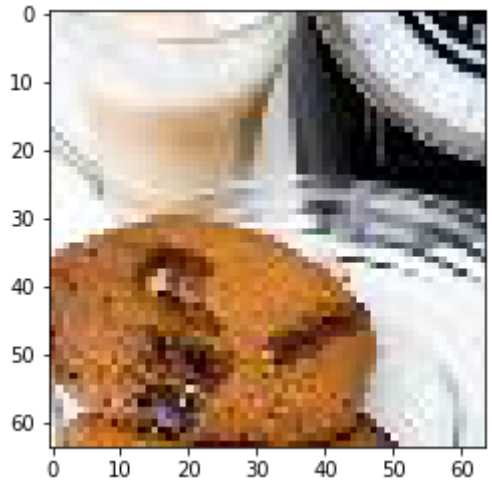
test 7810: Egyptian cat



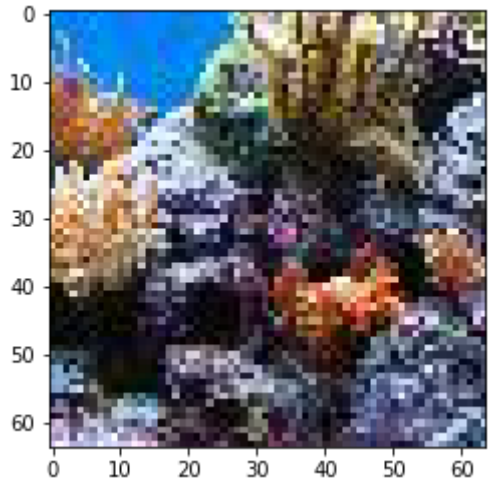
test 5597: snail



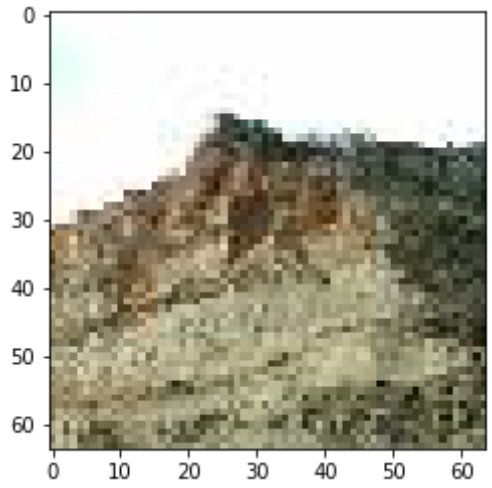
test 5693: plate



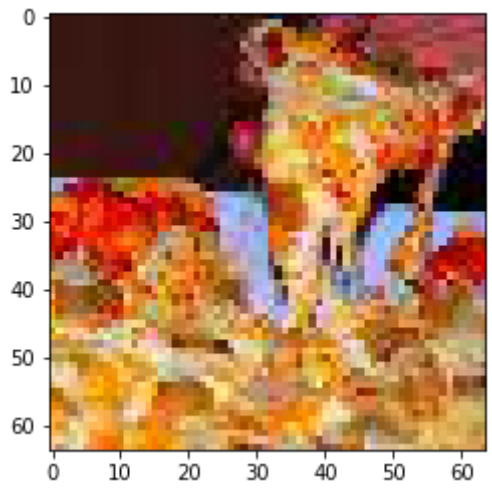
```
out of domain 737: coral reef
```



```
out of domain 4062: cliff, drop, drop-off
```



```
out of domain 1223: pizza, pizza pie
```



Make and Train Model

Implement a convolutional neural network similar to the one in [1]. We will simplify the architecture a bit, since we are dealing with Tiny Imagenet here, and since we would have trouble training the original model in Colab:

1. For the first convolutional layer, use a kernel size of 4 and stride 1, but still 96 filters.
2. For the *hidden* fully connected layers, use 2048 units, instead of 4096.

Otherwise, use the same architecture as in [1].

Some hints and remarks:

- Use 'same' padding for all convolutional and pooling layers.
- For the last layer, use *num_train_classes* (defined above) units.
- For all layers use *relu* activation functions, except for the last layer, where you should use the *softmax* activation.
- Apply dropout with dropout rate 0.5 before the two hidden fully connected layers.
- Train the model with the Adam optimizer, using a learning rate of 0.0001 and set *amsgrad=True*.
- Use early stopping [2] by calling model.fit(...) with argument *callbacks=[early_stopping_callback]*. The early_stopping_callback is already defined below. You'll need to provide the validation set as argument *validation_data* in model.fit(...).
- Train using *crossentropy* loss. You can use losses.sparse_categorical_crossentropy, since labels are not encoded in one-hot encoding.
- Use a *batch size* of 100.
- Train for maximal 100 epochs (early stopping will likely stop training much earlier).
- During training, measure *accuracy* and *top-5 accuracy*. You can use *sparse_top_k_categorical_accuracy*, since labels are not encoded in one-hot encoding.

[2] https://en.wikipedia.org/wiki/Early_stopping (https://en.wikipedia.org/wiki/Early_stopping)

In [0]:
```
def make_model():

  # keras model
  model = Sequential()

  # layer 1
  model.add(Conv2D(96, 4, padding='same', input_shape=(64, 64, 3)))
  model.add(MaxPooling2D((3,3), strides=2, padding='same'))
  model.add(ReLU())

  # layer 2
  model.add(Conv2D(192, 5, padding='same'))
  model.add(MaxPooling2D((3,3), strides=2, padding='same'))
  model.add(ReLU())

  # layer 3
  model.add(Conv2D(288, 3, padding='same'))
  model.add(ReLU())

  # layer 4
  model.add(Conv2D(288, 3, padding='same'))
  model.add(ReLU())

  # layer 5
  model.add(Conv2D(256, 3, padding='same'))
  model.add(MaxPooling2D((3,3), strides=2, padding='same', name='l5'))
  model.add(ReLU())

  # layer 6
  model.add(Flatten())
  model.add(Dropout(0.5))
  model.add(Dense(2048, name='l6'))
  model.add(ReLU())

  # layer 7
  model.add(Dropout(0.5))
  model.add(Dense(2048, name='l7'))
  model.add(ReLU())

  # layer 8 (output)
  model.add(Dense(num_train_classes, activation='softmax'))

  return model

def make_model_and_train():

  if not os.path.isdir(base_path):
    raise AssertionError('No folder base_path. Please run cell "Mount google drive" above.')

  model = make_model()
  model.summary()

  early_stopping_callback = keras.callbacks.EarlyStopping(monitor='val_loss',
                                                          min_delta=0,
                                                          patience=0,
                                                          verbose=1,
                                                          mode='auto')

  # compile model
  model.compile(optimizer=optimizers.Adam(learning_rate=0.0001, amsgrad=True),
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy', 'sparse_top_k_categorical_accuracy'])

  # train
  model.fit(train_x, train_y,
            batch_size = 100,
            epochs = 100,
            callbacks = [early_stopping_callback],
            validation_data = (val_x, val_y))

  # save model to google drive
  model.save(os.path.join(base_path, 'model.h5'))

  tf.keras.backend.clear_session()

make_model_and_train()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 96)        4704
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 32, 96)        0
_________________________________________________________________
re_lu (ReLU)                 (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 192)       460992
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 192)       0
_________________________________________________________________
re_lu_1 (ReLU)               (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 288)       497952
_________________________________________________________________
re_lu_2 (ReLU)               (None, 16, 16, 288)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 288)       746784
_________________________________________________________________
re_lu_3 (ReLU)               (None, 16, 16, 288)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 256)       663808
_________________________________________________________________
l5 (MaxPooling2D)            (None, 8, 8, 256)         0
_________________________________________________________________
re_lu_4 (ReLU)               (None, 8, 8, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 16384)             0
_________________________________________________________________
dropout (Dropout)            (None, 16384)             0
_________________________________________________________________
l6 (Dense)                   (None, 2048)              33556480
_________________________________________________________________
re_lu_5 (ReLU)               (None, 2048)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 2048)              0
_________________________________________________________________
l7 (Dense)                   (None, 2048)              4196352
_________________________________________________________________
re_lu_6 (ReLU)               (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 190)               389310
=================================================================
Total params: 40,516,382
Trainable params: 40,516,382
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
760/760 [==============================] - 93s 123ms/step - loss: 4.9332 - accuracy: 0.0329 - sparse_top_k_categorica
l_accuracy: 0.1115 - val_loss: 4.4373 - val_accuracy: 0.0809 - val_sparse_top_k_categorical_accuracy: 0.2425
Epoch 2/100
760/760 [==============================] - 95s 125ms/step - loss: 4.1850 - accuracy: 0.1134 - sparse_top_k_categorica
l_accuracy: 0.3019 - val_loss: 3.8667 - val_accuracy: 0.1615 - val_sparse_top_k_categorical_accuracy: 0.3812
Epoch 3/100
760/760 [==============================] - 96s 127ms/step - loss: 3.6874 - accuracy: 0.1861 - sparse_top_k_categorica
l_accuracy: 0.4214 - val_loss: 3.4776 - val_accuracy: 0.2243 - val_sparse_top_k_categorical_accuracy: 0.4660
Epoch 4/100
760/760 [==============================] - 97s 127ms/step - loss: 3.3616 - accuracy: 0.2393 - sparse_top_k_categorica
l_accuracy: 0.4971 - val_loss: 3.2394 - val_accuracy: 0.2676 - val_sparse_top_k_categorical_accuracy: 0.5231
Epoch 5/100
760/760 [==============================] - 97s 128ms/step - loss: 3.1102 - accuracy: 0.2838 - sparse_top_k_categorica
l_accuracy: 0.5522 - val_loss: 3.0981 - val_accuracy: 0.2914 - val_sparse_top_k_categorical_accuracy: 0.5540
Epoch 6/100
760/760 [==============================] - 97s 128ms/step - loss: 2.8849 - accuracy: 0.3238 - sparse_top_k_categorica
l_accuracy: 0.5995 - val_loss: 2.9681 - val_accuracy: 0.3125 - val_sparse_top_k_categorical_accuracy: 0.5802
Epoch 7/100
760/760 [==============================] - 97s 128ms/step - loss: 2.6742 - accuracy: 0.3626 - sparse_top_k_categorica
l_accuracy: 0.6432 - val_loss: 2.9076 - val_accuracy: 0.3242 - val_sparse_top_k_categorical_accuracy: 0.5961
Epoch 8/100
760/760 [==============================] - 97s 128ms/step - loss: 2.4727 - accuracy: 0.4014 - sparse_top_k_categorica
l_accuracy: 0.6853 - val_loss: 2.8294 - val_accuracy: 0.3420 - val_sparse_top_k_categorical_accuracy: 0.6132
Epoch 9/100
760/760 [==============================] - 97s 128ms/step - loss: 2.2655 - accuracy: 0.4426 - sparse_top_k_categorica
l_accuracy: 0.7226 - val_loss: 2.7864 - val_accuracy: 0.3548 - val_sparse_top_k_categorical_accuracy: 0.6183
Epoch 10/100
760/760 [==============================] - 97s 128ms/step - loss: 2.0659 - accuracy: 0.4787 - sparse_top_k_categorica
l_accuracy: 0.7607 - val_loss: 2.8423 - val_accuracy: 0.3508 - val_sparse_top_k_categorical_accuracy: 0.6226
Epoch 00010: early stopping
```

Question: Name two techniques which would likely improve the test accuracy.

We believe that the regularization of parameters of each layer would help increase the test accuracy, as it significantly reduces the variance of the model and decreases overfitting. This reduction helps the model to generalize better and achieve better accuracy on out-of-domain images. Moreover, we could tune the hyperparameters of our model, namely batch size, learning rate, number of filters of each convolutional layer, dropout rate, max-pooling size, etc. This could be done through a standard grid search (evaluate each combination and pick the highest) or a more nuanced approach, for example, Bayesian optimization (explore specific parts of the hyper-parameter space according to the expected improvement constructed through Bayesian techniques).

Evaluate Model

Evaluate the crossentropy, classification accuracy and top-5 classification accuracy on the train, validation and test sets.

In [0]:
```
import pandas as pd

model = load_model(os.path.join(base_path, 'model.h5'))

# evaluate model here
evaluations = {}
print('evaluating ...', flush=True)
evaluations['train'] = model.evaluate(train_x, train_y, verbose=0)
evaluations['validation'] = model.evaluate(val_x, val_y, verbose=0)
evaluations['test'] = model.evaluate(test_x, test_y, verbose=0)

pd.DataFrame(evaluations, ['cross_entropy', 'accuracy', 'top-5 accuracy'])
```

```
evaluating ...
```

Out[0]:

	train	validation	test
cross_entropy	1.335383	2.842271	2.948960
accuracy	0.679908	0.350842	0.338000
top-5 accuracy	0.887750	0.622632	0.601789

Image Retrieval

We are now using the trained model for image retrieval on the out-of-domain dataset ood_x. We are considering, in turn, single images from *ood_x* as query image, and the remaining 4,999 images as retrieval database. The task of image retrieval (IR) is to find the *K* most similar images to the query image. In [1], similarity is defined as Euclidean distance between l2-normalised neural codes, where neural codes are simply the outputs of particular ConvNet layers.

For each of the *3* layers which were considered in [1], perform the following steps:

1. Perform image retrieval for the first *10* images from *ood_x*. Retrieve the *K=5* most similar images for each query. Show the query image and the retrieved images next to each other, and mark the retrieved images which have the same class (stored in *ood_y*) as the query image. See Fig. 2 and 3 in [1] for examples (your results do not need to look precisely like in the paper. E.g., you can use *imshow* and *subplot*, and simply use *print* for the labels).
2. Compute and report the *mean average precision* (mAP), by computing the *average precision* (AP) for each image in *ood_x*, and taking the mean AP over all 5,000 images.

Hints:

- Make sure that the model is properly loaded, see previous cell.
- To obtain the neural codes, you can use the provided function eval_layer_batched(model, layer_name, ood_x, 100). It evaluates the layer with name *layer_name* for the whole ood_x. For example, if the layer contains 1024 units, this function will return a numpy array of size 5000x1024, with rows corresponding to images and columns to units.
- The AP is defined as follows.
  - Let *TP* be the number of *true positives*, that is, the number of retrieved images which have the *same* label as the query image.
  - Let *FP* be the number of *false positives*, that is, the number of retrieved images which have a *different* label than the query image.
  - Let *FN* be the number of *false negatives*, that is, the number of *non-retrieved* images, which have the *same* label as the query image.
  - The *precision* of an IR algorithm is defined as *precision* := TP / (TP + FP).
  - The *recall* is defined as *recall* := TP / (TP + FN).
  - To better understand precision and recall, figure a haystack with some needles in it. Precision will be high if you carefully select very few objects, where you are sure that these are needles. But recall will be low then. Recall will be high if you just grab and return the whole haystack. But precision will be low then. Thus, precision and recall are (usually) opposed to each other and represent a trade-off.
  - This trade-off can typically be governed by some hyper-parameter, in our case *K*, the number of retrieved images. For large *K*, we have large recall but low precision, for small *K* we have higher precision but low recall.
  - The trade-off can be inspected by looking at the precision-recall curve. The AP is defined as area under the precision-recall curve.
  - Fortunately, an estimator of AP is already implemented for you in the function *average_precision*. It takes two arguments:
    - sorted_class_vals: list of **class values** of the 4,999 other images, sorted according to closeness to the query image (closest first, most distant last).
    - true_class: the class values of the query image.

In [0]:

```
def get_layer_functor(model, layer_name):
  inp = model.input
  output =  model.get_layer(layer_name).output
  return K.function([inp], [output])

def eval_layer(x, layer_functor):
  return layer_functor(x)[0]

def eval_layer_batched(model, layer_name, x, batch_size):
  layer_functor = get_layer_functor(model, layer_name)
  idx = 0
  ret_vals = None
  while idx < x.shape[0]:
    if idx + batch_size > x.shape[0]:
      batch_x = x[idx:, ...]
    else:
      batch_x = x[idx:(idx+batch_size), ...]

    batch_vals = eval_layer(batch_x, layer_functor)
    if ret_vals is None:
      ret_vals = batch_vals
    else:
      ret_vals = np.concatenate((ret_vals, batch_vals), 0)

    idx += batch_size
  return ret_vals

def average_precision(sorted_class_vals, true_class):
  ind = sorted_class_vals == true_class
  num_positive = np.sum(ind)
  cum_ind = np.cumsum(ind).astype(np.float32)
  enum = np.array(range(1, len(ind)+1)).astype(np.float32)
  return np.sum(cum_ind * ind / enum) / num_positive

def show_single_img(img):
  plt.imshow(img)
  plt.axis("off")
  plt.show()

def show_row_of_imgs(imgs, labels, gt):
  n_of_imgs = len(imgs)
  fig, axes = plt.subplots(1, n_of_imgs, figsize=(10,10))

  for i, axis in enumerate(axes):
    axis.imshow(imgs[i])
    if labels[i] == gt:
      axis.patch.set_edgecolor('lightgreen')
      axis.patch.set_linewidth('15')
    axis.set_xticks([]), axis.set_yticks([])
  plt.show()

# variables
layers = ['l5', 'l6', 'l7']
samples = ood_x[:10]
labels = ood_y[:10]
nn_codes = [eval_layer_batched(model, layer, ood_x, 100) for layer in layers]
nn_codes[0] = nn_codes[0].reshape((5000, 8*8*256))
nn_codes_norm = [np.asarray([row / np.linalg.norm(row) for row in layer_codes]) for layer_codes in nn_codes]

# retrieve 5 most similar images
for i, sample in enumerate(samples):
  print(f'=== img {i}: {class_dict[labels[i]][:-1]}  ===')
  show_single_img(sample)

  for layer, layer_codes in zip(layers, nn_codes_norm):
    dist = np.linalg.norm(layer_codes - layer_codes[i], axis=1)
    indexes = dist.argsort()[1:6]
    print(f'Image retrieval on layer {layer}')
    show_row_of_imgs(ood_x[indexes], ood_y[indexes], labels[i])
```

=== img 0: coral reef ===

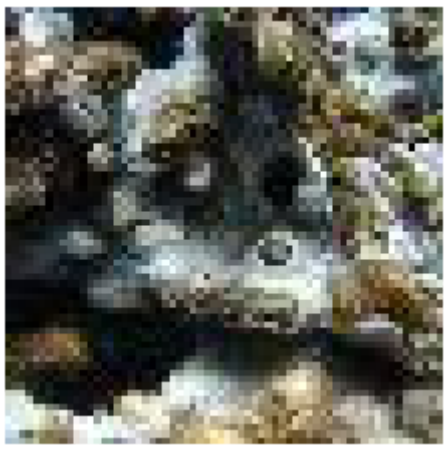


Image retrieval on layer l5

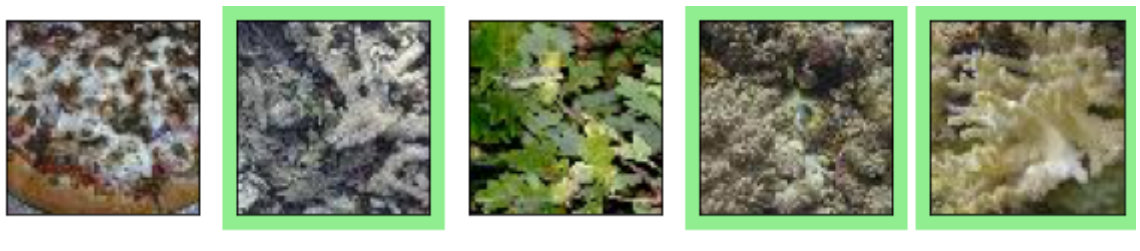


Image retrieval on layer l6

Image retrieval on layer l7

=== img 1: potpie ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 2: espresso ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 3: espresso ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 4: espresso ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 5: espresso ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 6: acorn ===

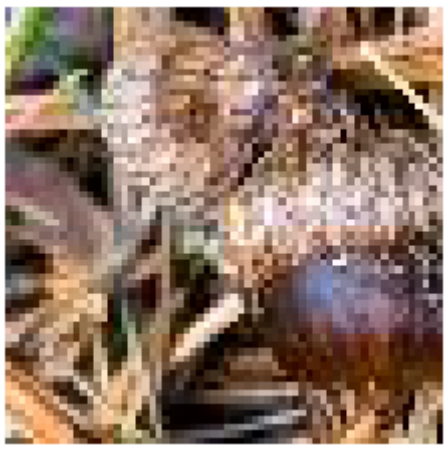


Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7

=== img 7: lakeside, lakeshore ===

Image retrieval on layer l5

Image retrieval on layer l6

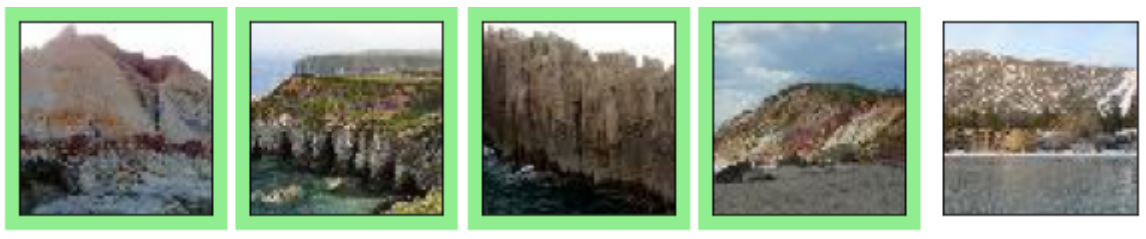
Image retrieval on layer l7

=== img 8: cliff, drop, drop-off ===

Image retrieval on layer l5

Image retrieval on layer l6

Image retrieval on layer l7



=== img 9: seashore, coast, seacoast, sea-coast ===

Image retrieval on layer l5

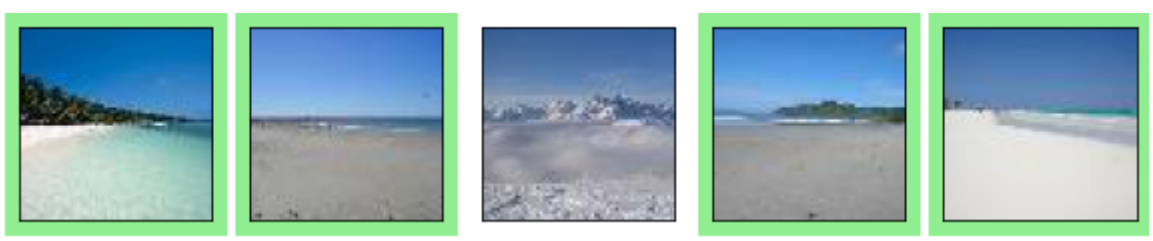


Image retrieval on layer l6

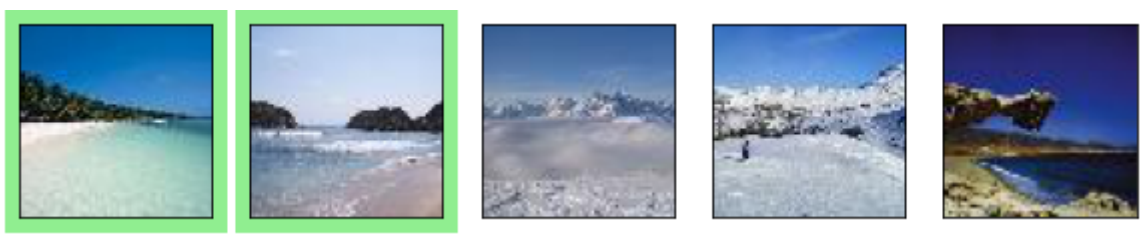


Image retrieval on layer l7

In [0]:
```
# mAP
for layer, layer_codes in zip(layers, nn_codes_norm):
  mAP = 0
  for i, label in enumerate(ood_y):
    dist = np.linalg.norm(layer_codes - layer_codes[i], axis=1)
    sort_idx = dist.argsort()[1:]
    mAP += average_precision(np.take(ood_y, sort_idx), ood_y[i])
  print(f'mAP for layer {layer} : {mAP/4999}')
```

```
mAP for layer l5 : 0.2828216071611716
mAP for layer l6 : 0.32304760753428635
mAP for layer l7 : 0.3446135221243652
```

Question: What are the qualitative differences between the different layers for neural codes?

Analyzing the images we retrieved on the out-of-domain dataset, we notice that there are differences between the images retrieved according to neural codes of different layers. It would appear that lower layers return neural codes that are more descriptive of the texture of the image, and therefore the retrieved images are images that have more similar textures. As an example, image 7 of class “acorn” shows similarly textured images being retrieved for the neural codes extracted by layer 5, although they are not of the correct class (out of 5 images retrieved, 4 are evidently pizzas). In contrast, neural codes extracted by higher levels would encode a more semantic meaning of the images, as they are the result of further learning on lower-level feature maps. Again considering the example of image 7, the pizzas that were retrieved according to the neural codes of layer 5 are now less close according to the Euclidean distance of l2-normalised layer 7 neural codes, with only one still being considered close.

Question: Do the observed mAP values (roughly) confirm the observations by Babenko et al.?

According to Babenko et al., the sixth layer should be the one that returns the best neural codes in terms of mAP. Due to differences in the network structure and the training and evaluation dataset, we can not directly compare our results with the ones in the paper. However, the relative ranking of the layers’ performances is roughly preserved. Layer 7 and 6 score around 0.33 mAP while layer 5 reaches only 0.28 mAP.