Open in Colab
(https://colab.research.google.com/github/Joriswillems/deeplearning/blob/master/assignment2/2IMM10_Assi

Assignment 2, Question 1

Group 82

- **Student 1** : Joris Willems + 0908753
- **Student 2** : Lars Schilders + 0908729

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 [150]:

```
%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
```

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 [2]:

```
# if not os.path.isdir('drive'):
#   drive.mount('drive')
# else:
#   print('drive already mounted')

# base_path = os.path.join('drive', 'My Drive', 'deeplearning2020_ass2_task1')
# if not os.path.isdir(base_path):
#   os.makedirs(base_path)
```

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 [3]:

```
# 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')
```

```
Found folder tiny-imagenet-200
```

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 [4]:

```
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, 'image
s')
    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, 'image
s', 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 137.37462401390076 seconds.
(76000, 64, 64, 3)
(9500, 64, 64, 3)
(9500, 64, 64, 3)
(5000, 64, 64, 3)
```

Show Some Images

In [5]:

```
# 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')
```

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 [5]:

```
def make_model():

  # keras model
  model = Sequential()

  # Make your model here
  # ...
  model.add(Conv2D(96, kernel_size=(4,4), activation='relu', padding='same', inp
ut_shape=(64,64,3)))
  model.add(MaxPooling2D(padding='same'))
  model.add(Conv2D(192, kernel_size=(5,5), activation='relu', padding='same'))
  model.add(MaxPooling2D(padding='same'))
  model.add(Conv2D(288, kernel_size=(3,3), activation='relu', padding='same'))
  model.add(Conv2D(288, kernel_size=(3,3), activation='relu', padding='same'))
  model.add(Conv2D(256, kernel_size=(3,3), activation='relu', padding='same'))
  model.add(MaxPooling2D(padding='same'))
  model.add(Flatten(name="L5"))
  model.add(Dropout(0.5))
  model.add(Dense(2048, activation='relu', name='L6'))
  model.add(Dropout(0.5))
  model.add(Dense(2048, activation='relu', name='L7'))
  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 dri
ve" above.')

  model = make_model()
  model.summary()

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

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

  model.fit(train_x, train_y, batch_size=100, epochs=100, callbacks=early_stoppi
ng_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         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 192)       460992    
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 192)       0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 288)       497952    
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 288)       746784    
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 256)       663808    
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 256)         0         
_________________________________________________________________
L5 (Flatten)                 (None, 16384)             0         
_________________________________________________________________
dropout (Dropout)            (None, 16384)             0         
_________________________________________________________________
L6 (Dense)                   (None, 2048)              33556480  
_________________________________________________________________
dropout_1 (Dropout)          (None, 2048)              0         
_________________________________________________________________
L7 (Dense)                   (None, 2048)              4196352   
_________________________________________________________________
dense (Dense)                (None, 190)               389310    
=================================================================
Total params: 40,516,382
Trainable params: 40,516,382
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
760/760 [==============================] - 158s 208ms/step - loss:
4.9573 - accuracy: 0.0284 - sparse_top_k_categorical_accuracy: 0.104
8 - val_loss: 4.4715 - val_accuracy: 0.0761 - val_sparse_top_k_categ
orical_accuracy: 0.2275
Epoch 2/100
760/760 [==============================] - 159s 210ms/step - loss:
4.2022 - accuracy: 0.1119 - sparse_top_k_categorical_accuracy: 0.298
4 - val_loss: 3.8052 - val_accuracy: 0.1682 - val_sparse_top_k_categ
orical_accuracy: 0.3926
Epoch 3/100
760/760 [==============================] - 159s 209ms/step - loss:
3.6883 - accuracy: 0.1867 - sparse_top_k_categorical_accuracy: 0.421
8 - val_loss: 3.4654 - val_accuracy: 0.2242 - val_sparse_top_k_categ
orical_accuracy: 0.4722
Epoch 4/100
760/760 [==============================] - 159s 209ms/step - loss:
3.3623 - accuracy: 0.2391 - sparse_top_k_categorical_accuracy: 0.497
1 - val_loss: 3.2828 - val_accuracy: 0.2580 - val_sparse_top_k_categ
orical_accuracy: 0.5154
Epoch 5/100
760/760 [==============================] - 159s 209ms/step - loss:
3.1073 - accuracy: 0.2819 - sparse_top_k_categorical_accuracy: 0.552
4 - val_loss: 3.1452 - val_accuracy: 0.2880 - val_sparse_top_k_categ
orical_accuracy: 0.5415
```

```
Epoch 6/100
760/760 [==============================] - 159s 209ms/step - loss:
2.8836 - accuracy: 0.3248 - sparse_top_k_categorical_accuracy: 0.599
8 - val_loss: 3.0446 - val_accuracy: 0.3020 - val_sparse_top_k_categ
orical_accuracy: 0.5677
Epoch 7/100
760/760 [==============================] - 159s 209ms/step - loss:
2.6681 - accuracy: 0.3657 - sparse_top_k_categorical_accuracy: 0.644
5 - val_loss: 2.9502 - val_accuracy: 0.3191 - val_sparse_top_k_categ
orical_accuracy: 0.5859
Epoch 8/100
760/760 [==============================] - 159s 209ms/step - loss:
2.4533 - accuracy: 0.4048 - sparse_top_k_categorical_accuracy: 0.688
8 - val_loss: 2.8982 - val_accuracy: 0.3333 - val_sparse_top_k_categ
orical_accuracy: 0.5935
Epoch 9/100
760/760 [==============================] - 159s 209ms/step - loss:
2.2408 - accuracy: 0.4463 - sparse_top_k_categorical_accuracy: 0.728
6 - val_loss: 2.9349 - val_accuracy: 0.3336 - val_sparse_top_k_categ
orical_accuracy: 0.5986
Epoch 00009: early stopping
```

Evaluate Model

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

In [151]:

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

In [6]:

```
# evaluate model here
# ...
print("Training set: ")
loss, acc, top_5_acc = model.evaluate(train_x, train_y, batch_size=100)
print("Cross entropy: {:.3f}, accuracy: {:.3f}, top-5 accuracy: {:.3f}".format(l
oss, acc, top_5_acc))
print("----")
print("Validation set: ")
loss, acc, top_5_acc  = model.evaluate(val_x, val_y, batch_size=100)
print("Cross entropy: {:.3f}, accuracy: {:.3f}, top-5 accuracy: {:.3f}".format(l
oss, acc, top_5_acc))
print("----")
print("Test set: ")
loss, acc, top_5_acc = model.evaluate(test_x, test_y, batch_size=100)
print("Cross entropy: {:.3f}, accuracy: {:.3f}, top-5 accuracy: {:.3f}".format(l
oss, acc, top_5_acc))
# ...
```

```
Training set:
760/760 [==============================] - 53s 70ms/step - loss: 1.4
944 - accuracy: 0.6381 - sparse_top_k_categorical_accuracy: 0.8615
Cross entropy: 1.494, accuracy: 0.638, top-5 accuracy: 0.861
----
Validation set:
95/95 [==============================] - 7s 69ms/step - loss: 2.9349
- accuracy: 0.3336 - sparse_top_k_categorical_accuracy: 0.5986
Cross entropy: 2.935, accuracy: 0.334, top-5 accuracy: 0.599
----
Test set:
95/95 [==============================] - 7s 69ms/step - loss: 3.0108
- accuracy: 0.3293 - sparse_top_k_categorical_accuracy: 0.5867
Cross entropy: 3.011, accuracy: 0.329, top-5 accuracy: 0.587
```

Name two techniques which would likely improve the test accuracy.

ANSWER

We train and evaluate the network in terms of a classification task. The high training accuracy and low test/val accuracy indicates that the network was severely overfitting. Classification accuracy on the test set can be improved by reducing overfitting (e.g. by using regularisation methods such as L1/L2-regularisation, data augmentation, reduce network capacity, batch normalisation or increase dropout rates).

It must be noted that we target learning representations for image retrieval, rather than high classification accuracy. The paper suggests that degradation of efficiency of the neural codes is rather graceful after compressing these neural codes (e.g. by PCA-based compression or discriminative dimensionality reduction).

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.

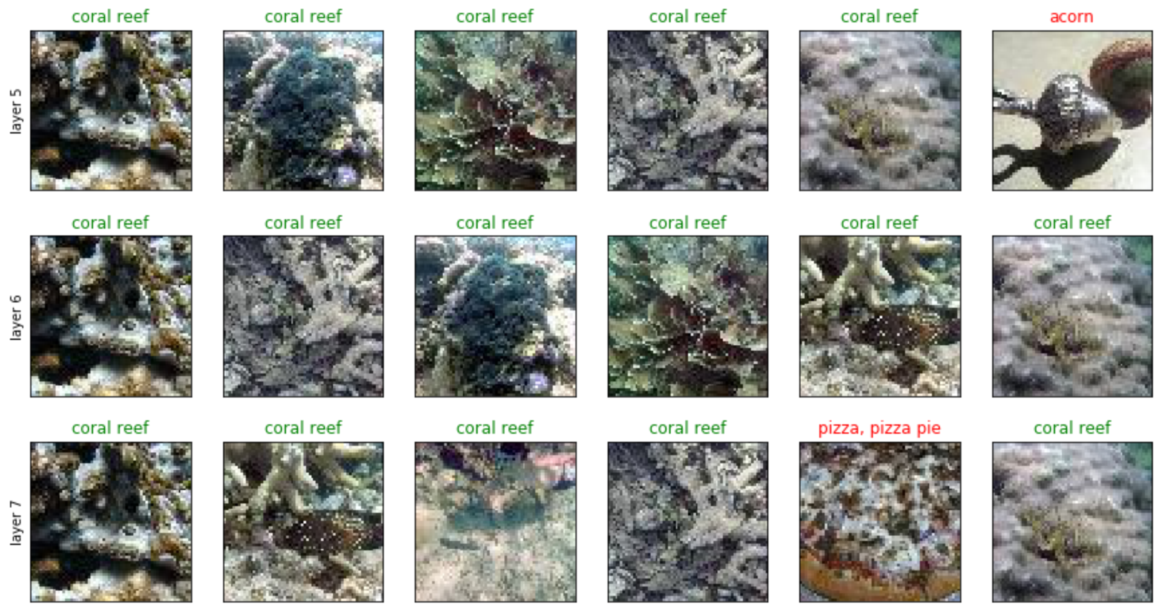
In [179]:

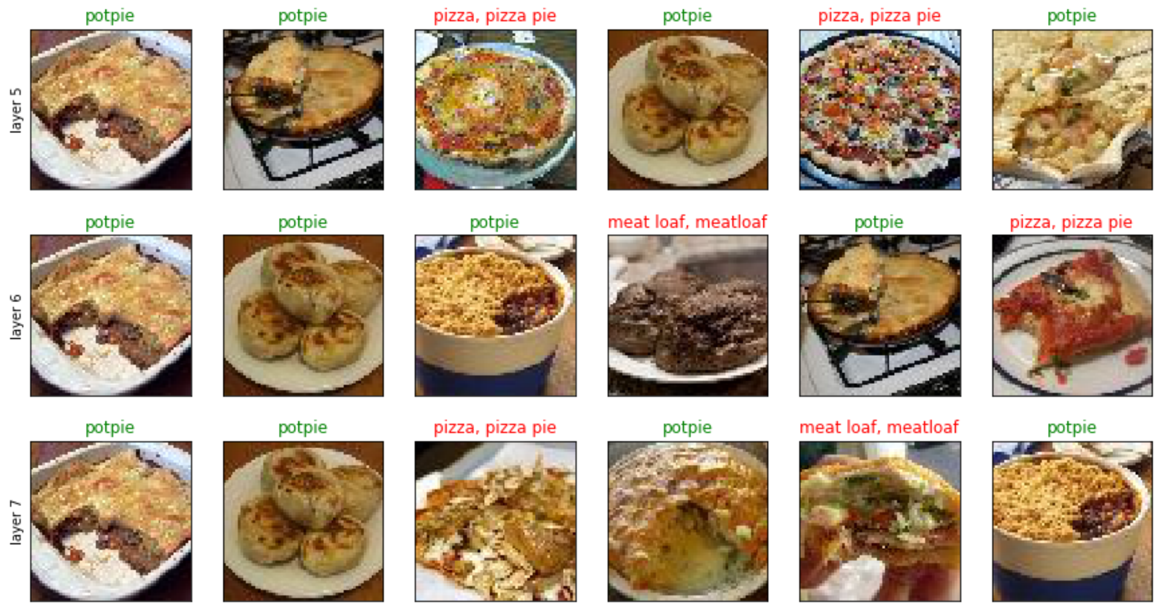
```
from sklearn.preprocessing import normalize
from sklearn.neighbors import NearestNeighbors

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


# perform image retrieval here
# ...
def get_kNN_indcs(layer_name, number_of_query_images=10, k_neighbors=5):
    neural_codes = normalize(eval_layer_batched(model, layer_name, ood_x, 100),
axis=1)
    knn = NearestNeighbors(n_neighbors=k_neighbors, n_jobs=-1, p=2)
    knn.fit(neural_codes)
    return knn.kneighbors(neural_codes[:number_of_query_images], n_neighbors=k_n
eighbors+1, return_distance=False)


def plot_layer_imgs(k):
    fig, axes = plt.subplots(nrows=3, ncols=6, figsize=(15,8))
    for i,axrow in enumerate(axes):
        for j, subplot in enumerate(axrow):
            idx = idcs[k, i,j]
            subplot.imshow(ood_x[idx])
            subplot.set_xticks([]); subplot.set_yticks([])
            subplot.set_ylabel("layer {}".format(i+5)) if j==0 else False
            color = 'green' if (ood_y[idx] == ood_y[k]) else 'red'
            subplot.set_title(class_dict.get(ood_y[idx])[:-1],  fontdict={'colo
r': color})
    print("\n\n")
    plt.show()


def get_MAP(layer_name):
    knn_idcs = get_kNN_indcs(layer_name, number_of_query_images=5000, k_neighbor
s=4999)[:, 1:]
    precision = [average_precision(ood_y[knn_idcs[sampleidx]], ood_y[sampleidx])
for sampleidx in range(len(knn_idcs))]
    return np.mean(precision)
```

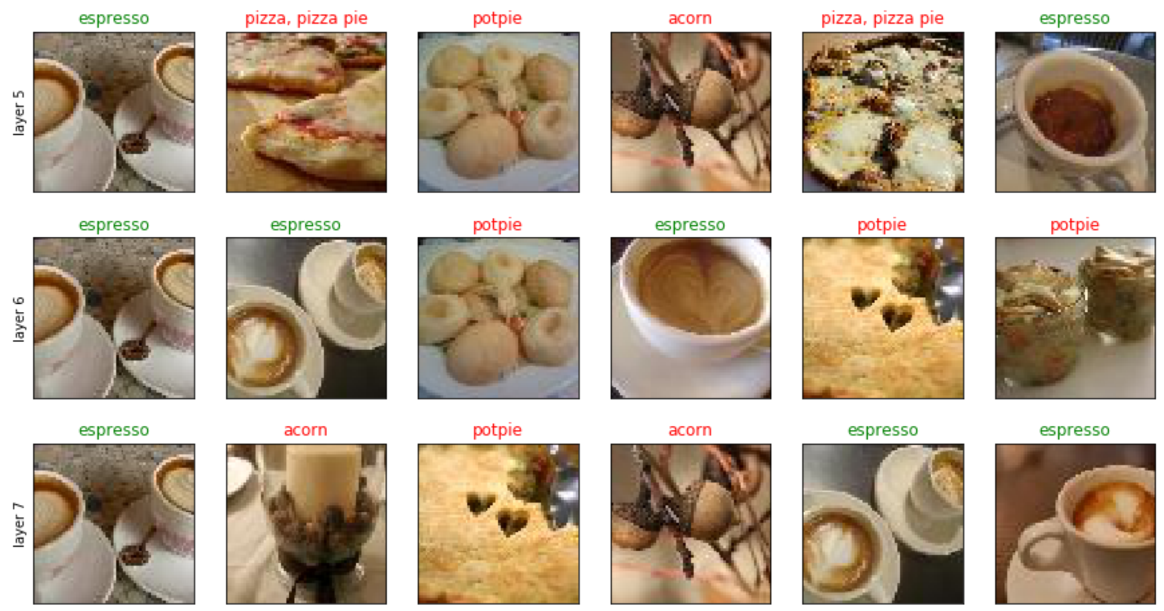
plot image retrieval

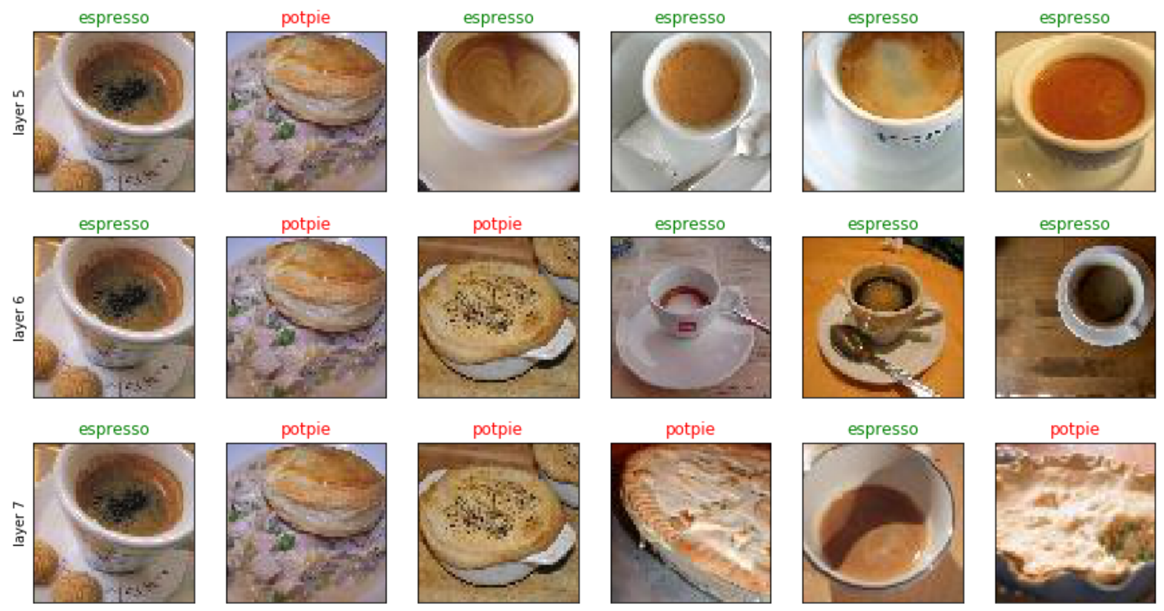
In [148]:

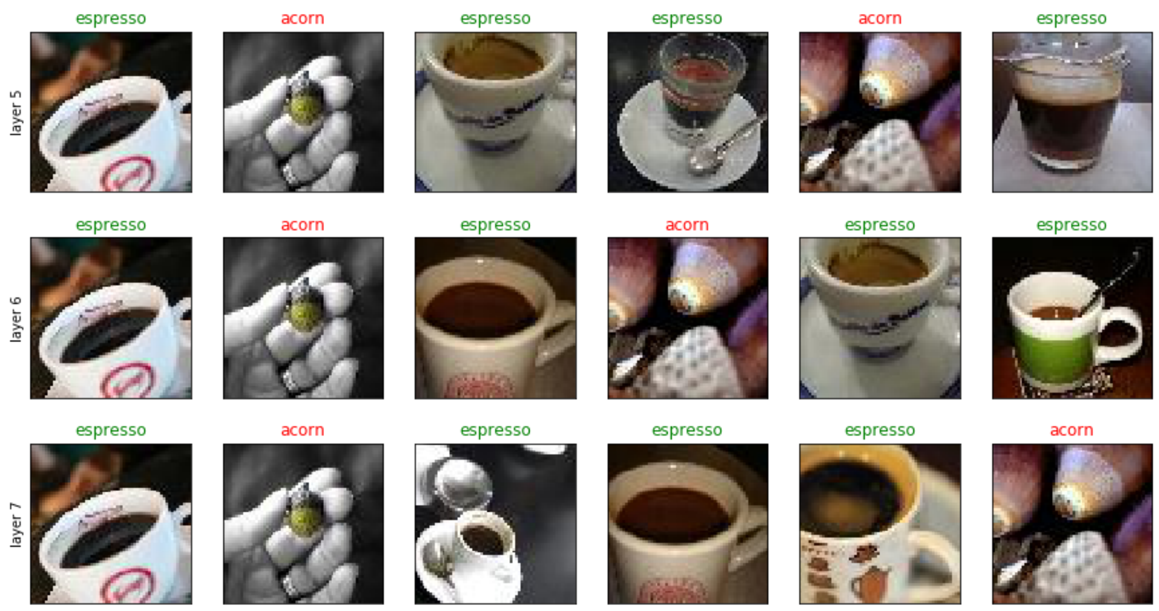
```
knn_indcs_5 = get_kNN_indcs("L5", number_of_query_images=10, k_neighbors=5)
knn_indcs_6 = get_kNN_indcs("L6", number_of_query_images=10, k_neighbors=5)
knn_indcs_7 = get_kNN_indcs("L7", number_of_query_images=10, k_neighbors=5)
idcs = np.swapaxes(np.concatenate([knn_indcs_5, knn_indcs_6, knn_indcs_7]).resha
pe((3, 10, 6)), axis1=0, axis2=1)
[plot_layer_imgs(i) for i in range(10)];
```

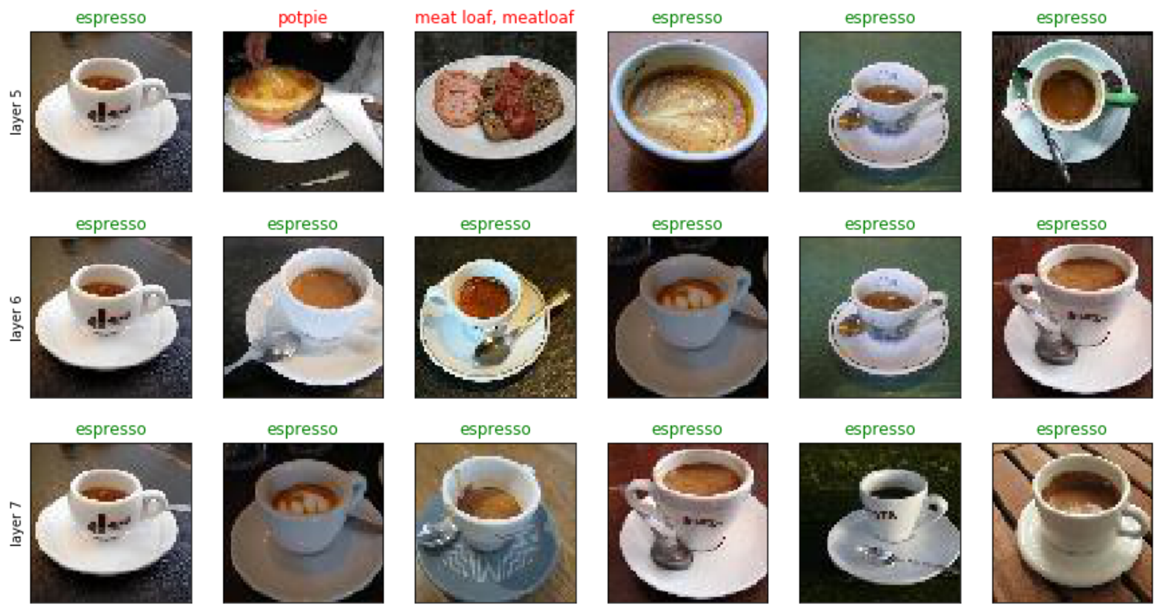

coral reef
coral reef
coral reef
coral reef
coral reef
acorn
layer 5
coral reef
coral reef
coral reef
coral reef
coral reef
coral reef
layer 6
coral reef
coral reef
coral reef
coral reef
pizza, pizza pie
coral reef
layer 7

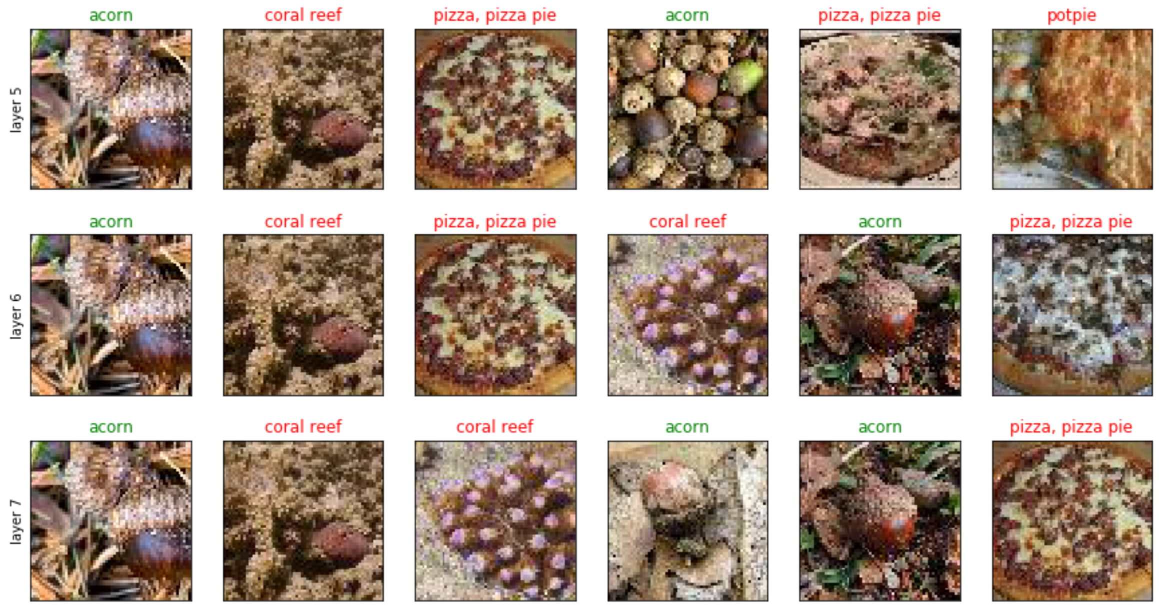

potpie
potpie
pizza, pizza pie
potpie
pizza, pizza pie
potpie
layer 5
potpie
potpie
potpie
meat loaf, meatloaf
potpie
pizza, pizza pie
layer 6
potpie
potpie
pizza, pizza pie
potpie
meat loaf, meatloaf
potpie
layer 7

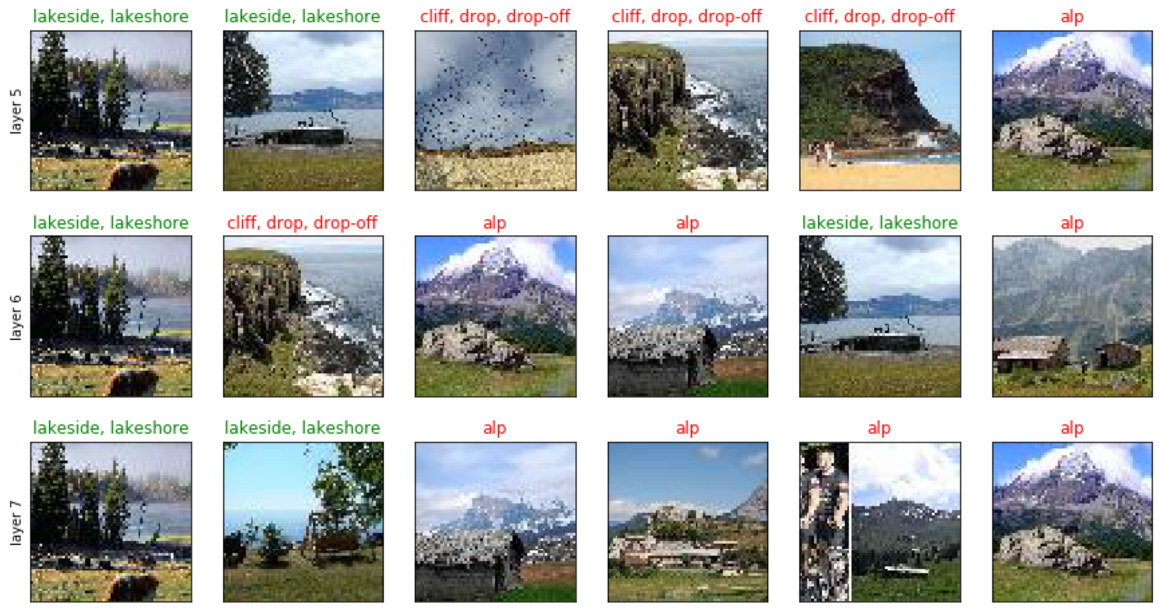

layer 5
espresso
pizza, pizza pie
potpie
acorn
pizza, pizza pie
espresso
layer 6
espresso
espresso
potpie
espresso
potpie
potpie
layer 7
espresso
acorn
potpie
acorn
espresso
espresso


layer 5
espresso
potpie
espresso
espresso
espresso
espresso
layer 6
espresso
potpie
potpie
espresso
espresso
espresso
layer 7
espresso
potpie
potpie
potpie
espresso
potpie


espresso
acorn
espresso
espresso
acorn
espresso
layer 5
espresso
acorn
espresso
acorn
espresso
espresso
layer 6
espresso
acorn
espresso
espresso
espresso
acorn
layer 7


espresso
potpie
meat loaf, meatloaf
espresso
espresso
espresso
layer 5
espresso
espresso
espresso
espresso
espresso
espresso
layer 6
espresso
espresso
espresso
espresso
espresso
espresso
layer 7


acorn
coral reef
pizza, pizza pie
acorn
pizza, pizza pie
potpie
layer 5
acorn
coral reef
pizza, pizza pie
coral reef
acorn
pizza, pizza pie
layer 6
acorn
coral reef
coral reef
acorn
acorn
pizza, pizza pie
layer 7


lakeside, lakeshore
lakeside, lakeshore
cliff, drop, drop-off
cliff, drop, drop-off
cliff, drop, drop-off
alp
layer 5
lakeside, lakeshore
cliff, drop, drop-off
alp
alp
lakeside, lakeshore
alp
layer 6
lakeside, lakeshore
lakeside, lakeshore
alp
alp
alp
alp
layer 7

precision accuracy

In [180]:

```
import pandas as pd

results= []
layer_names = ["L5", 'L6', 'L7']
[results.append(get_MAP(layer_name)) for layer_name in layer_names];

pd.DataFrame({'Mean precision' : results}, index=layer_names)
```

Out[180]:

	Mean precision
L5	0.291952
L6	0.322170
L7	0.345915

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

ANSWER

Our deep convolutional network learns spatial hierachies of patterns when trained on a classification task. More specifically, the first layers learns very local patterns (e.g. edges), while deeper layers learn specific combinations of patterns. The deepest layers learn the most complex abstractions and representations of the image data. Therefore, the neural codes of deeper layers are simply increasingly more complex representations of the initial image data. These representations can be used for image retrieval.

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

ANSWER

In Babenko et al. layer 6 performed consistently better, whereas for us layer 7 resulted in the best mAP value. A possible explanation could be that our image data is too complex to be accurately represented by neural codes from layers 5 and 6. The observations could be different when image retrieval is performed with images containing only simple shapes/patterns.

Furthermore, it must be noted that our layer 6 consists of 2048 neurons whereas the paper's architecture has 4096 neurons in layer 6, enabling more complex representations to be learned in the paper's layer 6. This could also explain the observed differences between our experiment and the paper's results.

Peer review

We worked together on all tasks, therefore, points can be distributed equally.

In [ ]:

```

```