

```
In [1]: import os
import time
import numpy as np

In [2]: def load_embeddings(embeddings_path):
    embeddings = {}
    with open(embeddings_path, 'r') as file:
        for line in file:
            splits = line.split()
            word = splits[0]
            coords = np.asarray(splits[1:], dtype='float32')
            embeddings[word] = coords
    return embeddings

In [3]: word_embeddings = load_embeddings('glove_6B_200d.txt')

In [4]: from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input

In [5]: def get_word_embedding(category, word_embeddings, wordvec_size=300):
    if category in word_embeddings:
        return word_embeddings[category]
    return np.zeros(shape=wordvec_size)

def preprocess_images(dataset_dir, word_embeddings, wordvec_size=300):
    image_list = []
    labels_list = []
    paths_list = []
    categories = os.listdir(dataset_dir)

    for category in categories:
        word_embedding = get_word_embedding(category, word_embeddings, wordvec_
image_names = os.listdir(os.path.join(dataset_dir, category))

        for img_name in image_names:
            full_path = os.path.join(dataset_dir, category, img_name)

            img_pil = image.load_img(full_path, target_size=(224, 224))
            img_raw = image.img_to_array(img_pil)
            img = preprocess_input(img_raw) # VGG16 image preprocessing.

            image_list.append(img)
            labels_list.append(word_embedding)
            paths_list.append(full_path)

    return np.array(image_list), np.array(labels_list), paths_list

In [6]: images_vgg16, image_embeddings, image_paths = preprocess_images('/dataset') w

In [7]: images_vgg16.shape

Out[7]: (1000, 224, 224, 3)

In [8]: from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dense, BatchNormalization, Activation, Dropout
from tensorflow.keras.losses import cosine_similarity

In [9]: def hybrid_model_backbone(intermediate_dim=2000, word_embedding_dim=300):
    vgg16 = VGG16()
    x = vgg16.get_layer('fc2').output

    for layer in vgg16.layers:
        layer.trainable = False
```

```

x = Dense(intermediate_dim, name="dense1")(x)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Dropout(0.5)(x)

x = Dense(intermediate_dim/2, name="dense2")(x)
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Dropout(0.5)(x)

x = Dense(word_embedding_dim, name="dense3")(x)
outputs = BatchNormalization()(x)

model = Model(inputs=[vgg16.input], outputs=outputs)
# https://faroit.com/keras-docs/2.0.8/optimizers/
sgd = optimizers.SGD(decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss=cosine_similarity)
return model

```

In [10]: hybrid\_model = hybrid\_model.backbone()

In [11]:

```

from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import load_model

```

In [12]:

```

def get_model_trained(model, X, y, model_path, train_model=False):
    if train_model:
        epochs, batch_size = (30, 32)
        X, y = shuffle(X, y, random_state=7)
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=0.2, random_state=7)
        checkpointer = ModelCheckpoint(
            filepath='best.hdf5', verbose=1, save_best_only=True)
        model.fit(X_train, y_train,
                  validation_data=(X_test, y_test),
                  epochs=epochs,
                  batch_size=batch_size,
                  callbacks=[checkpointer])
        model.save(model_path)
    return model
    # Assuming that there is a model in a given path.
    else:
        return load_model(model_path)

```

In [13]:

```

hybrid_model = get_model_trained(hybrid_model, images_vgg16, image_embeddings,
Epoch 1/30
25/25 [=====] - ETA: 0s - loss: -0.0110
Epoch 00001: val_loss improved from inf to -0.07970, saving model to best.hdf5
25/25 [=====] - 138s 6s/step - loss: -0.0110 - val_lo
ss: -0.0797
Epoch 2/30
25/25 [=====] - ETA: 0s - loss: -0.0509
Epoch 00002: val_loss improved from -0.07970 to -0.14033, saving model to bes
t.hdf5
25/25 [=====] - 146s 6s/step - loss: -0.0509 - val_lo
ss: -0.1403
Epoch 3/30
25/25 [=====] - ETA: 0s - loss: -0.0911
Epoch 00003: val_loss improved from -0.14033 to -0.20102, saving model to bes
t.hdf5
25/25 [=====] - 150s 6s/step - loss: -0.0911 - val_lo
ss: -0.2010
Epoch 4/30
25/25 [=====] - ETA: 0s - loss: -0.1240
Epoch 00004: val_loss improved from -0.20102 to -0.25002, saving model to bes
t.hdf5

```

In [14]:

```

from annoy import AnnoyIndex

```

```
In [15]: def get_annoy_index(vectors_list, number_of_trees=20, dims=300):
    annoy_index = AnnoyIndex(dims, 'angular')
    for i, vec in enumerate(vectors_list):
        annoy_index.add_item(i, vec)
    annoy_index.build(number_of_trees)
    return annoy_index

def generate_word_annoy_index(word_embeddings):
    word_index = dict(enumerate(word_embeddings))
    word_embeddings_list = [word_embeddings[word] for word in word_index.values]
    annoy_index = get_annoy_index(word_embeddings_list)
    return annoy_index, word_index
```

```
In [16]: # This might take minute or two (since it is a 400k corpus of 300d words).
word_annoy_index, word_index = generate_word_annoy_index(word_embeddings)
```

```
In [ ]:
```

```
In [17]: import json
```

```
In [18]: def generate_image_embeddings(image_paths, model):
    images = np.zeros(shape=images_vgg16.shape)
    image_path_index = dict(enumerate(image_paths))

    for ind, path in image_path_index.items():
        img_pil = image.load_img(path, target_size=images_vgg16.shape[1:-1])
        img_raw = image.img_to_array(img_pil)
        images[ind, :, :, :] = img_raw

    image_embeddings_list = model.predict(preprocess_input(images))
    return image_embeddings_list, image_path_index

def load_image_embeddings(vectors_filename, mapping_filename):
    image_embeddings = np.load("%s.npy" % vectors_filename)
    with open("%s.json" % mapping_filename) as f:
        image_path_index = json.load(f)
    return image_embeddings, {int(k): v for k, v in image_path_index.items()}

def save_image_embeddings(image_embeddings_filename,
                         image_embeddings,
                         mapping_filename,
                         image_path_index):
    np.save("%s.npy" % image_embeddings_filename, image_embeddings)
    with open("%s.json" % mapping_filename, 'w') as index_file:
        json.dump(image_path_index, index_file)
```

```
In [19]: def nearest_neighbors(vector, annoy_index, item_index, k=10):
    distances = annoy_index.get_nns_by_vector(vector, k, include_distances=True)
    return [[item_index[i], distances[i]] for i in range(k)]
```

```
In [ ]:
```

```
In [20]: def get_hybrid_embeddings(generate_embeddings=False):
    image_hybrid_embeddings_filepath = "./image_embeddings_30e3l"
    image_path_indexes_filepath = "./image_path_indexes_30e3l"
    # Generating embedding might take 5-10min.
    if generate_embeddings:
        image_hybrid_embeddings, image_path_index = \
            generate_image_embeddings(image_paths, hybrid_model)
        save_image_embeddings(image_hybrid_embeddings_filepath,
                             image_hybrid_embeddings,
                             image_path_indexes_filepath,
                             image_path_index)
    return image_hybrid_embeddings, image_path_index
else:
    image_hybrid_embeddings, image_path_index = \
        load_image_embeddings(image_hybrid_embeddings_filepath,
```

```

    image_path_indexes_filepath)
    return image_hybrid_embeddings, image_path_index
image_annoy_index = get_annoy_index(image_hybrid_embeddings, number_of_trees=10)

```

## DEMO

```

In [22]: from IPython.display import Image
from IPython.display import HTML, display
import urllib.request

# Assuming there are global variables (i.e. cells above did run):
# word_embeddings, word_annoy_index, word_index,
# hybrid_model, image_annoy_index, image_path_index

In [23]: def display_similar_images(mean_embedding, image_path=None):
    closest_images = nearest_neighbors(mean_embedding, image_annoy_index, image_path)

    html_str = "<script>$('div.cell.selected').next().height(100);</script>\n"
    if image_path is not None:
        html_str += "<h1>Our input image</h1>"
        html_str += "<img src='%s'>" % image_path
        html_str += "<h1>Similar images</h1>"
    html_str += "<table>

    for i in range(0, 5):
        left_cell = closest_images[2*i][1]
        html_str += "<tr><td><img src='%s'></td>" % left_cell
        right_cell = closest_images[2*i+1][1]
        html_str += "<td><img src='%s'></td></tr>" % right_cell
    html_str += "</table>\n"

    display(HTML(html_str))

def search_by_text(text):
    mean_embedding = np.mean([word_embeddings[word] for word in text.split()])
    display_similar_images(mean_embedding)

def get_image_labels(image_path, display_image=False):
    if display_image:
        display(Image(filename=image_path))

    images = np.zeros(shape=(1,) + images_vgg16.shape[1:])
    img = image.load_img(image_path, target_size=images_vgg16.shape[1:-1])
    x_raw = image.img_to_array(img)
    images[0] = np.expand_dims(x_raw, axis=0)

    inputs = preprocess_input(images)

    image_features = hybrid_model.predict(inputs)[0]
    closest_labels = nearest_neighbors(image_features, word_annoy_index, word_index)
    return closest_labels

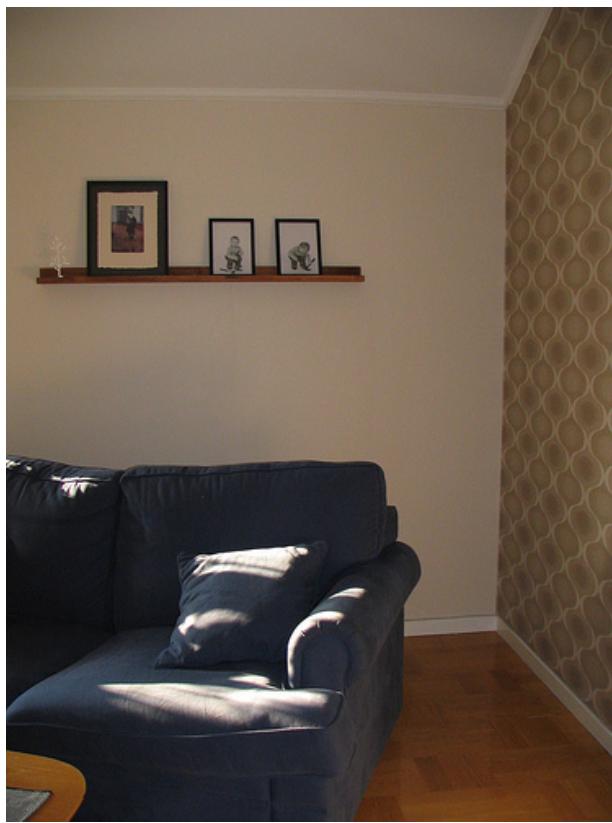
def search_similar_images(image_location, from_url=False):
    image_path = image_location
    if from_url:
        filename = image_location.split("/")[-1]
        urllib.request.urlretrieve(image_location, filename)
        image_path = filename

    closest_labels = [l for _, l, _ in get_image_labels(image_path)]
    mean_embedding = np.mean([word_embeddings[label] for label in closest_labels])
    display_similar_images(mean_embedding, image_path)

```

*Identifying labels on some pictures from the dataset:*

```
In [24]: get_image_labels(image_paths[222], display_image=True)
```



```
Out[24]: [[40518, 'ftaa', 1.2694400548934937],  
[14948, 'mat', 1.2847737073898315],  
[24861, 'clapton', 1.3002183437347412],  
[37737, 'overrule', 1.306251883506775],  
[23834, 'inked', 1.3084743022918701],  
[14167, 'alexei', 1.3096579313278198],  
[38733, 'ringside', 1.3123352527618408],  
[21269, 'primetime', 1.3125219345092773],  
[24510, 'cpa', 1.3148623704910278],  
[29601, 'nsc', 1.3152321577072144]]
```

In [25]: `get_image_labels(image_paths[262], display_image=True)`



Out[25]: [[8433, 'bicycle', 0.6421365141868591],  
[25032, 'motorbike', 0.6621052026748657],  
[7213, 'motorcycle', 0.7363784909248352],  
[569, 'car', 0.9455204606056213],  
[1907, 'vehicle', 0.9974915981292725],  
[2575, 'truck', 1.020522117614746],  
[8837, 'parked', 1.0407575368881226],  
[12238, 'jeep', 1.0540876388549805],  
[3069, 'drivers', 1.056908369064331],  
[2031, 'driving', 1.0696306228637695]]

*Searching for specific category/label:*

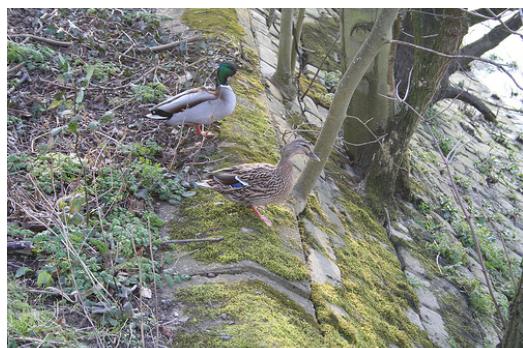
In [26]: `search_by_text("aeroplane")`





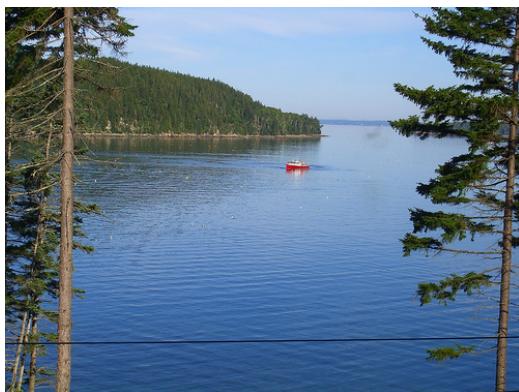
*Searching for categories that are not in image dataset:*

In [27]: `search_by_text("aviation")`





In [28]: `search_by_text("ocean")`





In [29]: `search_by_text("street")`





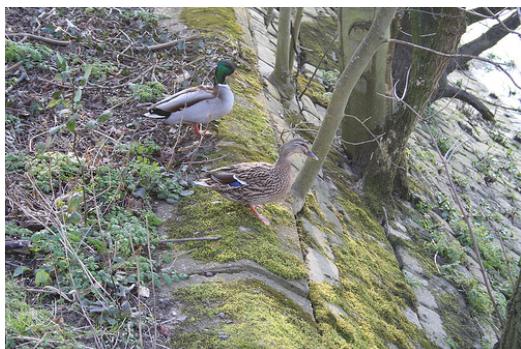
*Achieving more complex queries by combining words:*

In [30]: `search_by_text("horse_jockey")`





In [31]: `search_by_text("bird near water")`





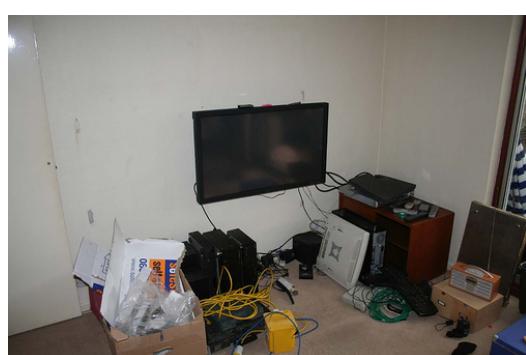
Since we are able to identify labels on an image, we can perform search by image:

```
In [32]: search_similar_images(image_paths[650])
```

## Our input image



## Similar images





```
In [33]: search_similar_images(image_paths[711])
```

## Our input image



## Similar images





The same goes for images outside of the dataset

In [34]: `search_similar_images(' /demo_images/modern_interior_design/home_preview.jpg')`

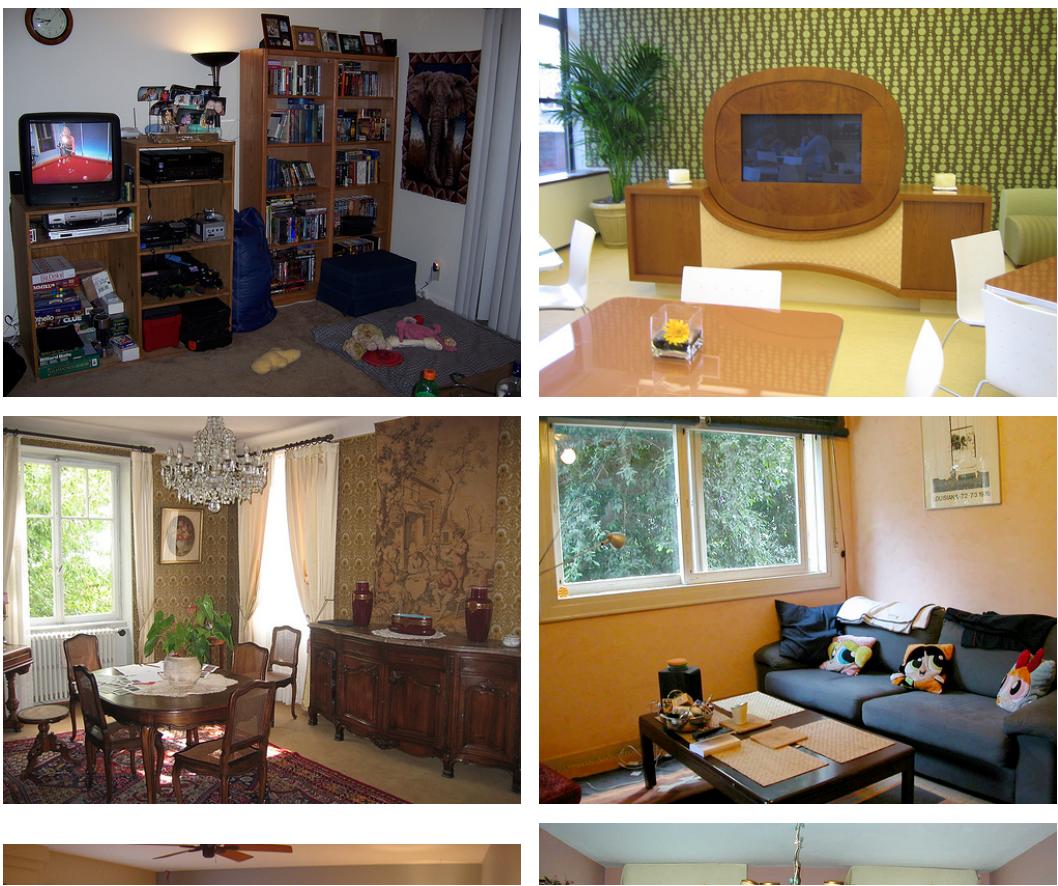
## Our input image





## Similar images





```
In [35]: search_similar_images('~/demo_images/death_railway_1000x1000_060_720.jpg')
```

## Our input image



## Similar images





```
In [36]: search_similar_images('./demo_images/moor-moorland-grasses-wetland-nature-conse
```

## Our input image





## Similar images

