Streamlit

Building a Semantic Search Engine

In this post, I will outline how to use vector representations to create a **search engine**.

We will go through three successive steps

- How to search for similar images to an input image
- How to search for similar words and synonyms
- Generating tags from images, and searching for images using language

To do this, we will use a simple approach of using **embeddings**, vector representations of images and text.Once we have embeddings, searching simply becomes a matter of finding vectors close to our input vector.

The way we do this is by calculating the **cosine distance** between our image embedding, and embeddings for other images. Similar images will have similar embeddings, meaning a **low cosine distance between embeddings**.

To start things off, we will need a dataset to experiment with.

Dataset

Loading the data

Let's start by loading our dataset, which consists of a total of a **1000 images**, divided in **20 classes** with 50 images for each.

This dataset can be found here. *Credit to Cyrus Rashtchian, Peter Young, Micah Hodosh, and Julia Hockenmaier.*

In addition, we load GloVe vectors pre-trained on Wikipedia, which we will use when we incorporate text.

```
word_vectors = vector_search.load_glove_vectors("models/glove.6B")
images, vectors, image_paths = load_paired_img_wrd('dataset', word_vectors)
```

Here is what our load function looks like:

```
def load_paired_img_wrd(folder, word_vectors, use_word_vectors=True):
    class_names = [fold for fold in os.listdir(folder) if ".DS" not in fold]
    image_list = []
    labels_list = []
    paths_list = []
    for cl in class_names:
        splits = cl.split("_")
        if use_word_vectors:
            vectors = np.array([word_vectors[split] if split in word_vectors else np.zeros(sh
            class_vector = np.mean(vectors, axis=0)
        subfiles = [f for f in os.listdir(folder + "/" + cl) if ".DS" not in f]

        for subf in subfiles:
            full_path = os.path.join(folder, cl, subf)
            img = image.load_img(full_path, target_size=(224, 224))
            x_raw = image.img_to_array(img)
            x_expand = np.expand_dims(x_raw, axis=0)
            x = preprocess_input(x_expand)
            image_list.append(x)
            if use_word_vectors:
                labels_list.append(class_vector)
            paths_list.append(full_path)
    img_data = np.array(image_list)
    img_data = np.rollaxis(img_data, 1, 0)
    img_data = img_data[0]

    return img_data, np.array(labels_list), paths_list
```

Here is a list of our classes: `aeroplane` `bicycle` `bird` `boat` `bottle` `bus` `car` `cat` `chair` `cow` `dining_table` `dog` `horse` `motorbike` `person` `potted_plant` `sheep` `sofa` `train` `tv_monitor`

We now have a tensor of images of size (1000, 224, 224, 3), of word vectors of size (1000, 300), and a list of corresponding file paths.

Visualizing the data

Let's see what our data looks like, here is one example image from each class

aeroplane

bicycle

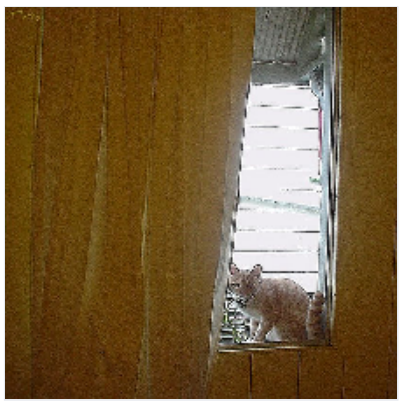
bird

boat

bottle

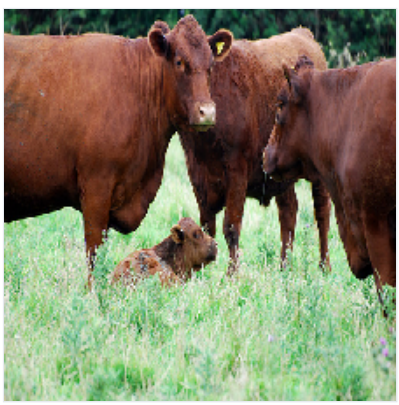
bus

car

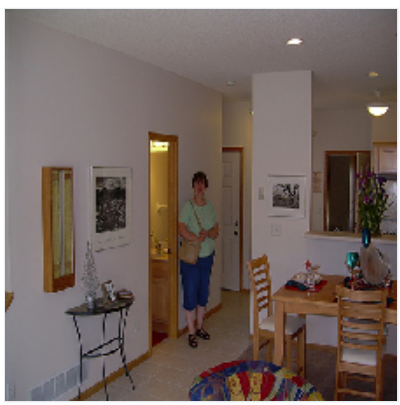


cat

chair

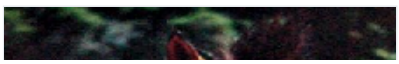


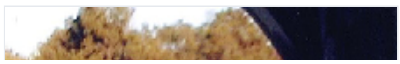
cow

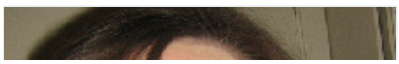


dining_table

dog







horse

motorbike

person

potted_plant

sheep

sofa

train

tv_monitor

We can see our labels are pretty **noisy**, many photos contain multiple categories, and the label is not always from the most prominent one.

Indexing the images

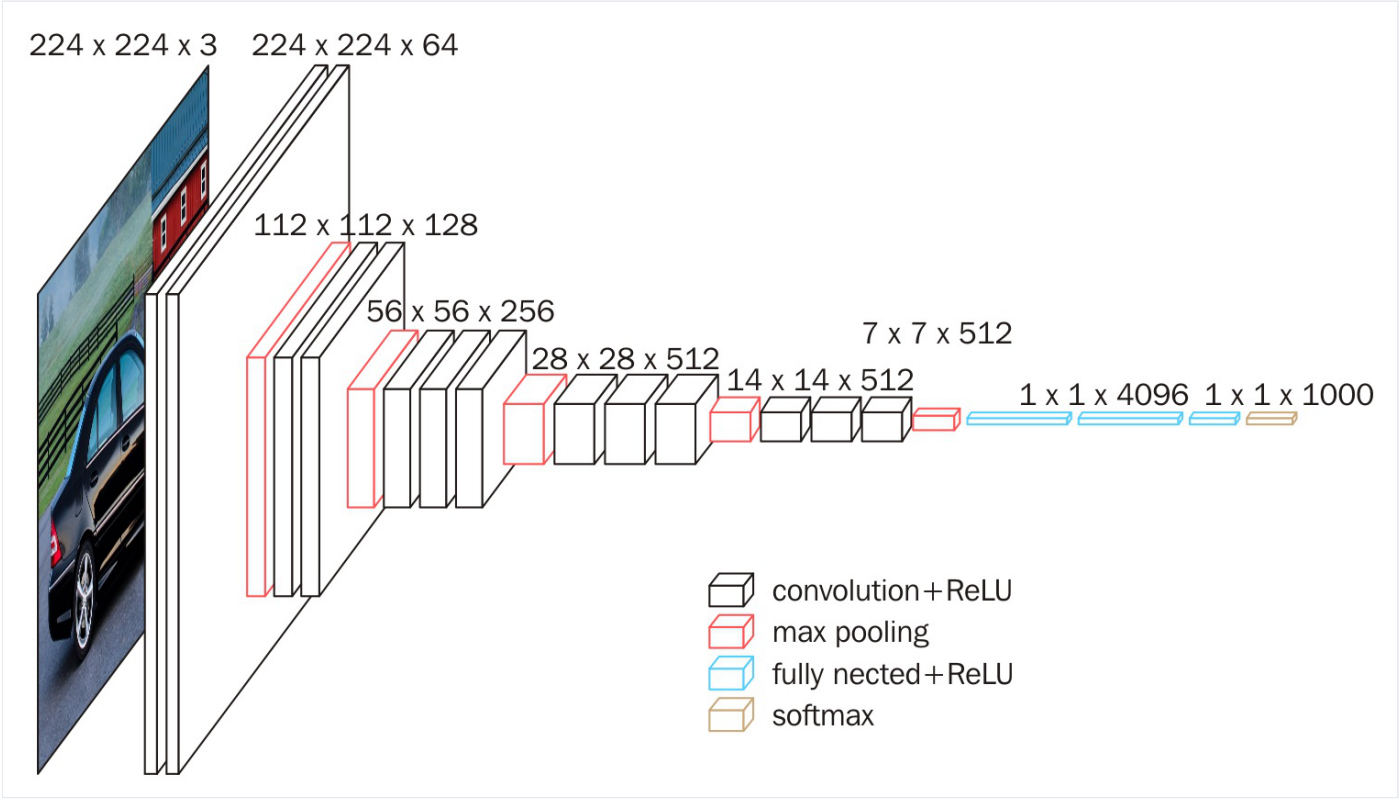
We are now going to load a model that was **pre-trained** on a large data set (imagenet), and is freely available online.

We use this model to generate **embeddings** for our images.

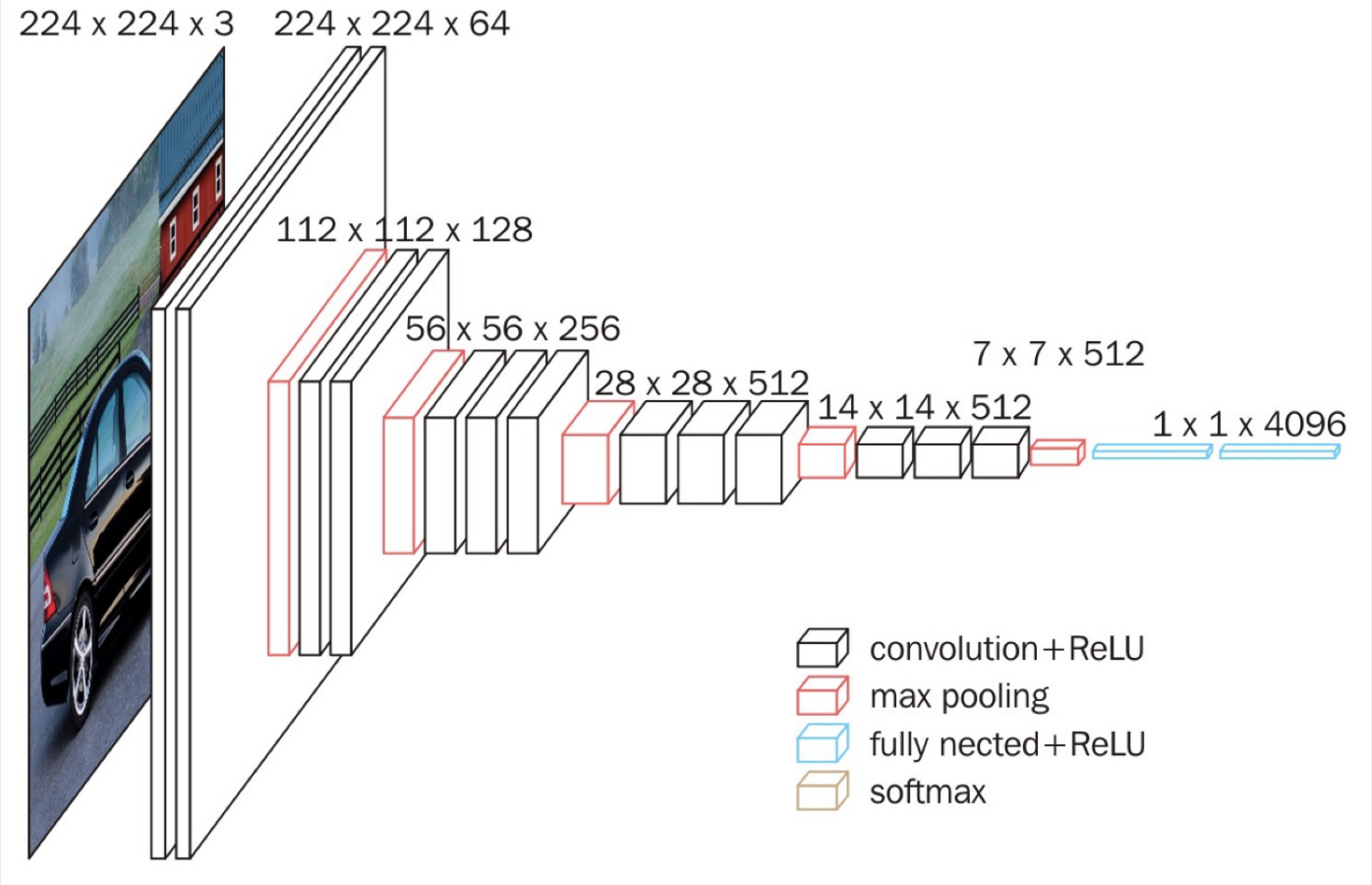
As you can see below, once we've used the model to generate image features, we can then **store them to disk** and re-use them without needing to do inference again! This is one of the reason that embeddings are so popular in practical applications, as they allow for huge efficiency gains.

```
model = vector_search.load_headless_pretrained_model()
if generate_image_features:
    images_features, file_index = vector_search.generate_features(image_paths, model)
    vector_search.save_features(features_path, images_features, file_mapping_path, file_index
else:
    images_features, file_index = vector_search.load_features(features_path, file_mapping_pat
```

Our model is simply VGG16 without the last layer (softmax)



Original VGG. Credit to Data Wow Blog



Our model

This is how we get such a model in practice

```
def load_headless_pretrained_model():
    """
    Loads the pretrained version of VGG with the last layer cut off
    :return: pre-trained headless VGG16 Keras Model
    """
    pretrained_vgg16 = VGG16(weights='imagenet', include_top=True)
    model = Model(inputs=pretrained_vgg16.input,
                  outputs=pretrained_vgg16.get_layer('fc2').output)
    return model
```

What do we mean by generating embeddings? Well we just use our pre-trained model up to the penultimate layer, and store the value of the activations.

```
def generate_features(image_paths, model):
    """
    Takes in an array of image paths, and a trained model.
    Returns the activations of the last layer for each image
    :param image_paths: array of image paths
    :param model: pre-trained model
    :return: array of last-layer activations, and mapping from array_index to file_path
    """
    start = time.time()
    images = np.zeros(shape=(len(image_paths), 224, 224, 3))
    file_mapping = {i: f for i, f in enumerate(image_paths)}

    # We load all our dataset in memory because it is relatively small
    for i, f in enumerate(image_paths):
        img = image.load_img(f, target_size=(224, 224))
        x_raw = image.img_to_array(img)
        x_expand = np.expand_dims(x_raw, axis=0)
        images[i, :, :, :] = x_expand

    logger.info("%s images loaded" % len(images))
    inputs = preprocess_input(images)
    logger.info("Images preprocessed")
    images_features = model.predict(inputs)
    end = time.time()
    logger.info("Inference done, %s Generation time" % (end - start))
    return images_features, file_mapping
```

Here are what the embeddings look like for the first 20 images. Each image is now represented by a sparse vector of size 4096:

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	4.3119	0	0	0	0	3.0562	0	0	0
1	0	0	0	0	3.3479	0	0	0	0	0	0
2	0	0	8.8749	0	0	1.1841	0.1892	1.6967	0	0	0
3	0	0	3.6080	0	0	0	0	0	1.8356	0	0
4	0	0	4.1273	0	2.1862	0	0	1.3936	0	3.9333	0
5	0	1.4552	1.5524	2.8287	3.9478	0	0	1.2723	0	0	0
6	0	0	6.4489	0.9040	0.0829	0	0.8447	0	0	0	0
7	0	1.4600	0	0	0	1.1291	0.5071	0	0	0	0	0.1
8	0	0	6.7299	0	0	1.3051	1.7395	4.2032	0	5.9994	0
9	0	0	4.7380	0.6831	1.2823	0	0	0	0	0	0
10	0.1210	0.7178	0	0.2367	0	0	0	0.2125	0	0	0

Now that we have the features, we will build a fast index to search through them using Annoy.

```
image_index = vector_search.index_features(images_features)
```

```
def index_features(features, n_trees=1000, dims=4096, is_dict=False):
    """
    Use Annoy to index our features to be able to query them rapidly
    :param features: array of item features
    :param n_trees: number of trees to use for Annoy. Higher is more precise but slower.
    :param dims: dimension of our features
    :return: an Annoy tree of indexed features
    """
    feature_index = AnnoyIndex(dims, metric='angular')
    for i, row in enumerate(features):
        vec = row
        if is_dict:
            vec = features[row]
        feature_index.add_item(i, vec)
    feature_index.build(n_trees)
    return feature_index
```

Using our embeddings to search through images

We can now simply take in an image, get its **embedding** (saved to disk), and look in our fast index to find **similar embeddings, and thus similar images**.

This is especially useful, since image labels are often noisy, so there is more to an image than it's label.

In our dataset for example, we have both a class `cat`, and a class `bottle`.

Which class do you think this image is labeled as?

Cat or bottle

The correct answer is **bottle** ... This is an actual issue that comes up often in real datasets. Labeling images as unique categories is quite limiting, which is why we hope to use more nuanced representations.

Luckily, this is exactly what deep learning is good at!

Let's see if our image search using embeddings does better than human labels

Searching for index `200`, file `dataset/bottle/2008_000112.jpg`

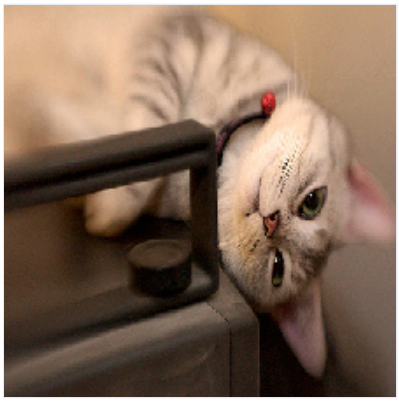
```
results = vector_search.search_index_by_key(search_key, image_index, file_index)
```

```
def search_index_by_key(key, feature_index, item_mapping, top_n=10):
    """
    Search an Annoy index by key, return n nearest items
    :param key: the index of our item in our array of features
    :param feature_index: an Annoy tree of indexed features
    :param item_mapping: mapping from indices to paths/names
    :param top_n: how many items to return
    :return: an array of [index, item, distance] of size top_n
    """
    distances = feature_index.get_nns_by_item(key, top_n, include_distances=True)
    return [[a, item_mapping[a], distances[1][i]] for i, a in enumerate(distances[0])]
```

- `[200, 'dataset/bottle/2008_000112.jpg', 8.953361975727603e-05]`
- `[375, 'dataset/cat/2008_004635.jpg', 0.9413783550262451]`
- `[399, 'dataset/cat/2008_007888.jpg', 0.9977052211761475]`
- `[361, 'dataset/cat/2008_002201.jpg', 1.0260729789733887]`
- `[778, 'dataset/potted_plant/2008_006068.jpg', 1.0308609008789062]`
- `[370, 'dataset/cat/2008_003622.jpg', 1.0327214002609253]`
- `[360, 'dataset/cat/2008_002067.jpg', 1.032860517501831]`
- `[220, 'dataset/bottle/2008_004795.jpg', 1.0681318044662476]`
- `[381, 'dataset/cat/2008_005386.jpg', 1.0685954093933105]`
- `[445, 'dataset/chair/2008_007941.jpg', 1.074022650718689]`

OG

0.9413783550262451



0.9977052211761475

1.0260729789733887

1.0308609008789062

1.0327214002609253

1.032860517501831

1.0681318044662476

1.0685954093933105

Great, we mostly get more images of **cats**, which seems very reasonable!. One image in there however, is of a shelf of bottles.

This approach perform wells to find similar images in general, but some times we are only interested in **part of the image**.

For example, given an image of a cat and a bottle, we might be only interested in similar cats, not similar bottles.

A common approach is to use an **object detection** model first, detect our cat, and do image search on a cropped version of the original image.

This adds a huge computing overhead, which we would like to avoid if possible.

There is a simpler "hacky" approach, which consists of **re-weighing** the activations of the last layer using our target class's weights. This cool trick initially brought to my attention by Insight Fellow Daweon Ryu

Let's demonstrate how this works, by weighing our activations according to class `284` in Imagenet, `Siamese cat`.

```
# Index 284 is the index for the Siamese cat class in Imagenet
weighted_features = vector_search.get_weighted_features(284, images_features)
weighted_index = vector_search.index_features(weighted_features)
weighted_results = vector_search.search_index_by_key(search_key, weighted_index, file_index)
```

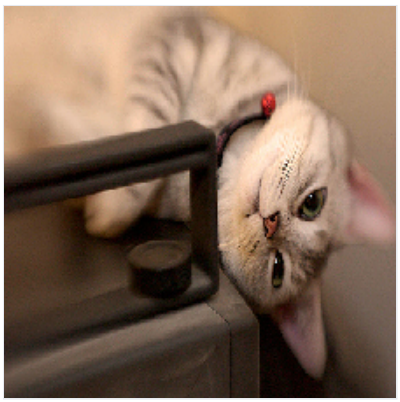
Here is how we perform our **trick** of re-weighing the features.

```
def get_weighted_features(class_index, images_features):
    """
    Use class weights to re-weigh our features
    :param class_index: Which Imagenet class index to weigh our features on
    :param images_features: Unweighted features
    :return: Array of weighted activations
    """
    class_weights = get_class_weights_from_vgg()
    target_class_weights = class_weights[:, class_index]
    weighted = images_features * target_class_weights
    return weighted
```

- [200, 'dataset/bottle/2008_000112.jpg', 0.00011064470891142264]
- [375, 'dataset/cat/2008_004635.jpg', 0.8694091439247131]
- [399, 'dataset/cat/2008_007888.jpg', 0.968299150466919]
- [370, 'dataset/cat/2008_003622.jpg', 0.9725740551948547]
- [361, 'dataset/cat/2008_002201.jpg', 1.007677435874939]
- [360, 'dataset/cat/2008_002067.jpg', 1.0096632242202759]
- [445, 'dataset/chair/2008_007941.jpg', 1.013108730316162]
- [778, 'dataset/potted_plant/2008_006068.jpg', 1.019992470741272]
- [812, 'dataset/sheep/2008_004621.jpg', 1.021153211593628]
- [824, 'dataset/sheep/2008_006327.jpg', 1.0285786390304565]

OG

0.8694091439247131

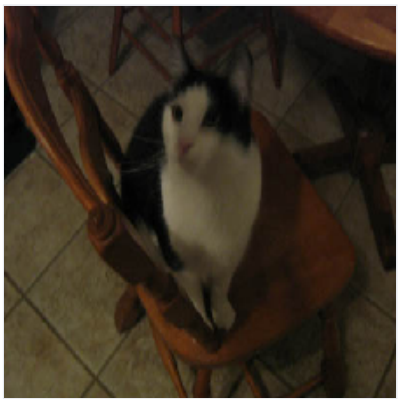


0.968299150466919

0.9725740551948547

1.007677435874939

1.0096632242202759



1.013108730316162

1.019992470741272

1.021153211593628

We can see that the search has been biased to look for **Siamese cat like things**. We no longer show any bottles, but we do return an image of a sheep for the last image, which is much more cat-like than bottle-like!

We have seen we can search for similar images in a **broad** way, or by **conditioning on a particular class** our model was trained on.

This is a great step forward, but since we are using a model **pre-trained on Imagenet**, we are thus limited to the thousand **Imagenet classes**. These classes are far far from all encompassing (they lack a category for human for example), so we would ideally like to find something more **flexible**

Shifting to words

Taking a detour to the world of NLP, could we use a similar approach to index and search for words?

We loaded a set of pre-trained vectors from GloVe, which were obtaining by crawling through all of Wikipedia, and learning the semantic relationships between words in that dataset.

Here is what our pre-trained word embeddings look like, dense vectors of size 300:

```
st.write("word", word_vectors["word"])
st.write("vector", word_vectors["vector"])
try:
    st.write("fwjwiwejiu", word_vectors["fwjwiwejiu"])
except KeyError as key:
    st.write("The word %s does not exist" % key)
```

word

	0
0	-0.4304
1	-0.4062
2	-0.1828
3	-0.1255
4	0.0435
5	-0.2122
6	-0.3651
7	0.0326
8	-0.1083
9	-1.3537
10	-0.1915

vector

	0
0	0.5912
1	0.9842
2	0.9996
3	-0.4865
4	-0.9500
5	-0.0678
6	0.3405
7	-0.3943
8	0.3806
9	-1.1546
10	-0.0009

The word 'fwjwiwejiu' does not exist

Indexing

Just like before, we will create an indexer, this time indexing all of the GloVe vectors

```
word_index, word_mapping = vector_search.build_word_index(word_vectors)
```

```
def build_word_index(word_vectors):
    """
    Builds a fast index out of a list of pretrained word vectors
    :param word_vectors: a list of pre-trained word vectors loaded from a file
    :return: an Annoy tree of indexed word vectors and a mapping from the Annoy index to the
    """
    logging.info("Creating mapping and list of features")
    word_list = [(i, word) for i, word in enumerate(word_vectors)]
    word_mapping = {k: v for k, v in word_list}
    word_features = [word_vectors[lis[1]] for lis in word_list]
    logging.info("Building tree")
    word_index = index_features(word_features, n_trees=20, dims=300)
    logging.info("Tree built")
    return word_index, word_mapping
```

Searching

Now, we can **search our embeddings** for similar words!

Searching for said

- `[16, 'said', 0.0]`
- `[154, 'told', 0.688713550567627]`
- `[391, 'spokesman', 0.7859575152397156]`
- `[476, 'asked', 0.872875452041626]`
- `[3852, 'noting', 0.9151610732078552]`
- `[1212, 'warned', 0.915908694267273]`
- `[2558, 'referring', 0.9276227951049805]`
- `[859, 'reporters', 0.9325974583625793]`
- `[2948, 'stressed', 0.9445104002952576]`
- `[171, 'tuesday', 0.9446316957473755]`

Searching for one

- `[48, 'one', 0.0]`
- `[170, 'another', 0.6205118894577026]`
- `[91, 'only', 0.666000247001648]`
- `[55, 'two', 0.6895312070846558]`
- `[58, 'first', 0.7803555130958557]`
- `[215, 'same', 0.7992483973503113]`
- `[79, 'time', 0.8085516691207886]`
- `[359, 'every', 0.8182190656661987]`
- `[120, 'just', 0.8255878686904907]`
- `[126, 'second', 0.8277111649513245]`

Searching for (

- `[23, '(', 0.0]`
- `[24, ')', 0.21370187401771545]`
- `[652, '8', 1.0106892585754395]`
- `[422, '5', 1.0143887996673584]`
- `[524, '6', 1.0183145999908447]`
- `[632, '7', 1.0220754146575928]`
- `[790, '9', 1.0285060405731201]`
- `[314, '3', 1.0295473337173462]`
- `[409, '4', 1.0337133407592773]`
- `[45, ':', 1.0341565608978271]`

This is a pretty general method, but our representations seem **incompatible**. The embeddings for images are of size 4096, while the ones for words are of size 300, how could we use one to search for the other?

In addition, even if both embeddings were the same size, they were each trained in a completely different fashion, so it is incredibly unlikely that images and related words would happen to have the same embeddings randomly. We need to train a **joint model**.

Worlds Collide

Creating a hybrid model

Let's now create a **hybrid** model that can go from words to images and vice versa.

Here, we are actually training our own model. The way we use this, is by drawing inspiration from a great paper called DeViSE.

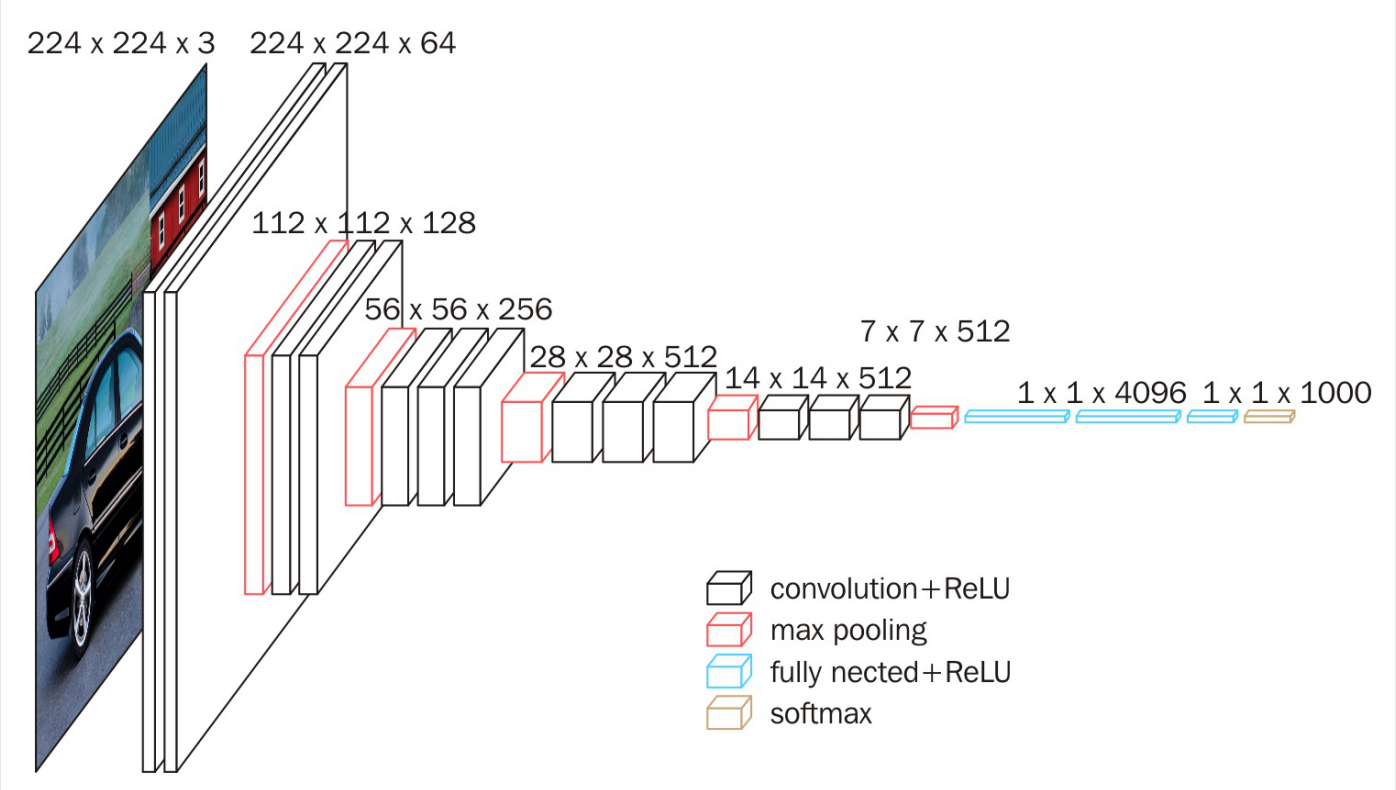
The idea is to combine both representations by re-training our image model and **change its labels**.

Usually, image classifiers are trained to pick a category out of many (1000 for Imagenet). What this translates to is that for Imagenet for example, the last layer is a vector of size 1000 representing the **probability of each class**. This is the layer we had previously removed.

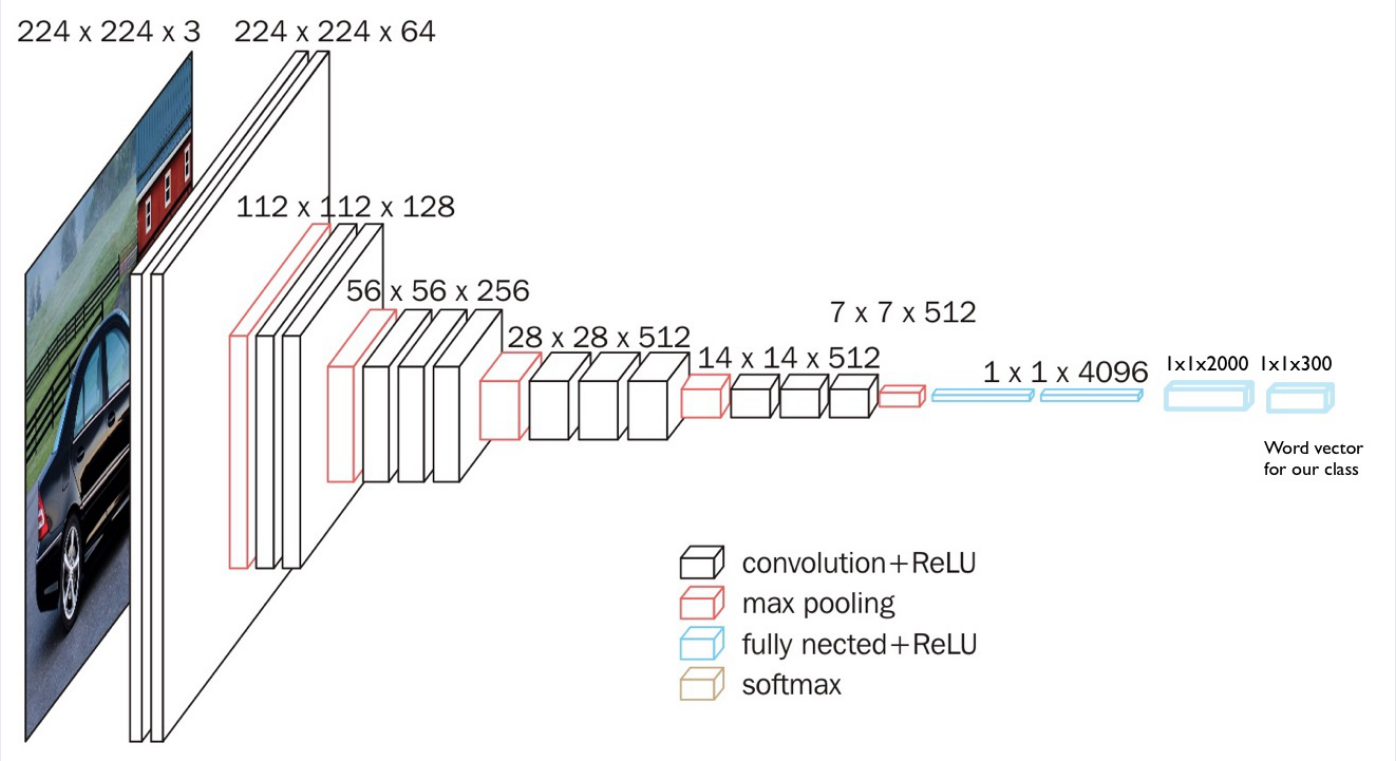
For this next part, we replace the target of our model with the **word vector of our category**. This allows our model to learn to map the semantics of images to the semantics of words!

Here, we have added two dense layers, leading to an output layer of size 300.

Here is what the model looked like when it was trained on Imagenet:



Here is what it looks like now:



Training the model

And here is the code to build it.

```
custom_model = vector_search.setup_custon_model()
```

```
def setup_custon_model(intermediate_dim=2000, word_embedding_dim=300):
    """
    Builds a custom model taking the fc2 layer of VGG16 and adding two dense layers on top
    :param intermediate_dim: dimension of the intermediate dense layer
    :param word_embedding_dim: dimension of the final layer, which should match the size of c
    :return: a Keras model with the backbone frozen, and the upper layers ready to be trained
    """
    headless_pretrained_vgg16 = VGG16(weights='imagenet', include_top=True, input_shape=(224,
    x = headless_pretrained_vgg16.get_layer('fc2').output

    # We do not re-train VGG entirely here, just to get to a result quicker (fine-tuning the
    # lead to better results)
    for layer in headless_pretrained_vgg16.layers:
        layer.trainable = False

    image_dense1 = Dense(intermediate_dim, name="image_dense1")(x)
    image_dense1 = BatchNormalization()(image_dense1)
    image_dense1 = Activation("relu")(image_dense1)
    image_dense1 = Dropout(0.5)(image_dense1)

    image_dense2 = Dense(word_embedding_dim, name="image_dense2")(image_dense1)
    image_output = BatchNormalization()(image_dense2)

    complete_model = Model(inputs=[headless_pretrained_vgg16.input], outputs=image_output)
    sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    complete_model.compile(optimizer=sgd, loss=cosine_proximity)
    return complete_model
```

We then re-train our model on a training split of our dataset, to learn to predict **the word_vector associated with the label of an image**.

For an image with the category cat for example, we are trying to predict the 300-length vector associated with cat.

This training takes a bit of time, so I saved a model I trained on my laptop overnight.

It is important to note that the training data we are using here (80% of our dataset, so 800 images) is minuscule, compared to usual datasets (Imagenet has **a million** images, **3 orders of magnitude more**). If we were using a traditional technique of training with categories, we would not expect our model to perform incredibly well on the test set, and would certainly not expect it to perform well on completely new examples.

```
if train_model:
    with st.echo():
        num_epochs = 50
        batch_size = 32
        st.write(
            "Training for %s epochs, this might take a while, "
            "change train_model to False to load pre-trained model" % num_epochs)
        x, y = shuffle(images, vectors, random_state=2)
        X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state
        checkpointer = ModelCheckpoint(filepath='checkpoint.hdf5', verbose=1, save_best_only=
        custom_model.fit(X_train, y_train, validation_data=(X_test, y_test),
                         epochs=num_epochs, batch_size=batch_size, callbacks=[checkpointer])
        custom_model.save(model_save_path)
else:
    st.write("Loading model from `%s`" % model_load_path)
    custom_model = load_model(model_load_path)
```

Loading model from `cpu_5_hours.hdf5`

Building two indices (words and images)

Our model is trained and ready

For our word index, we will use the one we built previously.

Now, to build a fast image index, we need to run a forward pass on every image with our new model

```
if generate_custom_features:
    hybrid_images_features, file_mapping = vector_search.generate_features(image_paths, custo
    vector_search.save_features(custom_features_path, hybrid_images_features, custom_features
                                file_mapping)
else:
    hybrid_images_features, file_mapping = vector_search.load_features(custom_features_path,
                                                                       custom_features_file_m
image_index = vector_search.index_features(hybrid_images_features, dims=300)
```

Here are what our embeddings look like now for the first 20 images

They are of length 300, just **like our word vectors**!

	0	1	2	3	4	5	6	7	8	9
0	0.8140	-0.9699	-0.8195	-1.0473	-2.8413	-0.7623	1.6234	0.6338	0.7516	1.5667
1	0.8622	-0.2703	-0.2305	-0.6366	-0.5915	-0.2352	-0.2312	-0.1933	-0.0867	-1.0736
2	0.5945	-0.5109	-0.5081	-0.1498	-2.3880	-1.0420	0.8915	1.0736	-0.0201	1.5657
3	0.3023	0.1107	-0.8508	-1.2166	-1.8913	-0.7346	0.3182	0.8193	0.2075	0.6533
4	0.2118	-0.1245	0.0768	-0.5300	-2.1813	-0.6441	0.4625	0.7159	-0.0474	0.6065
5	0.4480	-0.5513	-0.4355	-0.5590	-2.7080	-1.2363	1.2703	0.5861	0.3877	0.4058
6	1.2390	-1.0059	-0.6487	-0.6985	-3.7131	-2.2962	1.8469	1.5147	0.4059	2.5356
7	0.2433	-1.7297	-0.2811	-1.2918	-2.0497	-0.4008	1.7059	0.7052	0.9428	-0.5898
8	0.6367	-1.2549	-1.0381	-0.3851	-4.3205	-2.2462	1.6842	1.1207	0.6268	3.0632
9	0.1041	-0.0840	-0.8556	-0.3975	-2.3832	-0.7906	0.7822	0.4879	0.2993	0.3289
10	0.7036	-1.9394	-0.0356	-1.1400	-1.3237	0.1400	1.7112	0.4523	0.7689	-0.6306

Generating semantic tags

We can now easily extract tags from any image

Let's try with our cat/bottle image

Generating tags for `dataset/bottle/2008_000112.jpg`

```
results = vector_search.search_index_by_value(hybrid_images_features[200], word_index, word_r
```

```
def search_index_by_value(vector, feature_index, item_mapping, top_n=10):
    """
    Search an Annoy index by value, return n nearest items
    :param vector: the index of our item in our array of features
    :param feature_index: an Annoy tree of indexed features
    :param item_mapping: mapping from indices to paths/names
    :param top_n: how many items to return
    :return: an array of [index, item, distance] of size top_n
    """
    distances = feature_index.get_nns_by_vector(vector, top_n, include_distances=True)
    return [[a, item_mapping[a], distances[1][i]] for i, a in enumerate(distances[0])]
```

- `[5450, 'cat', 0.8766047954559326]`
- `[18990, 'litter', 1.173615574836731]`
- `[39138, 'raccoon', 1.1843332052230835]`
- `[11408, 'monkey', 1.1853549480438232]`
- `[18389, 'leopard', 1.1956979036331177]`
- `[22454, 'puppy', 1.2021284103393555]`
- `[18548, 'sunglasses', 1.2086421251296997]`
- `[13748, 'goat', 1.2140742540359497]`
- `[22718, 'squirrel', 1.217861294746399]`
- `[8891, 'mask', 1.2214369773864746]`

These results are reasonable, let's try to see if we can detect more than the bottle in the messy image below.

Generating tags for `dataset/bottle/2008_005023.jpg`

```
results = vector_search.search_index_by_value(hybrid_images_features[223], word_index, word_r
```

- `[6676, 'bottle', 0.3879561722278595]`
- `[7494, 'bottles', 0.7513495683670044]`
- `[12780, 'cans', 0.9817070364952087]`
- `[16883, 'vodka', 0.9828150272369385]`
- `[16720, 'jar', 1.0084964036941528]`
- `[12714, 'soda', 1.0182772874832153]`
- `[23279, 'jars', 1.0454961061477661]`
- `[3754, 'plastic', 1.0530102252960205]`
- `[19045, 'whiskey', 1.061428427696228]`
- `[4769, 'bag', 1.0815287828445435]`

The model learns to extract **many relevant tags**, even from categories that it was not trained on!

Searching for images using text

Most importantly, we can use our joint embedding to search through our image database using any word. We simply need to get our pre-trained word embedding from GloVe, and find the images that have the most similar embeddings! Generalized image search with minimal data.

Let's start first with a word that was actually in our training set

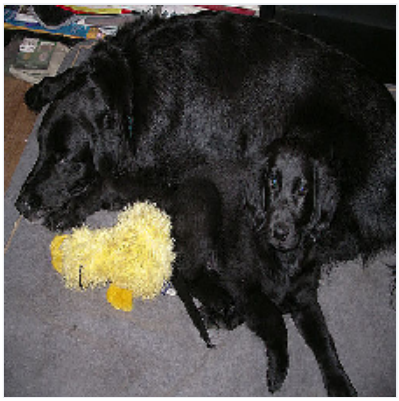
```
results = vector_search.search_index_by_value(word_vectors["dog"], image_index, file_mapping)
```

- `[578, 'dataset/dog/2008_004931.jpg', 0.3864878714084625]`
- `[589, 'dataset/dog/2008_005890.jpg', 0.3922061026096344]`
- `[567, 'dataset/dog/2008_002859.jpg', 0.4144909679889679]`
- `[568, 'dataset/dog/2008_003133.jpg', 0.4158168137073517]`
- `[555, 'dataset/dog/2008_000706.jpg', 0.4299579858779907]`
- `[590, 'dataset/dog/2008_006130.jpg', 0.43139806389808655]`
- `[576, 'dataset/dog/2008_004760.jpg', 0.4334023892879486]`
- `[556, 'dataset/dog/2008_000808.jpg', 0.43694719672203064]`
- `[561, 'dataset/dog/2008_001479.jpg', 0.43990498781204224]`
- `[574, 'dataset/dog/2008_004653.jpg', 0.44838857650756836]`

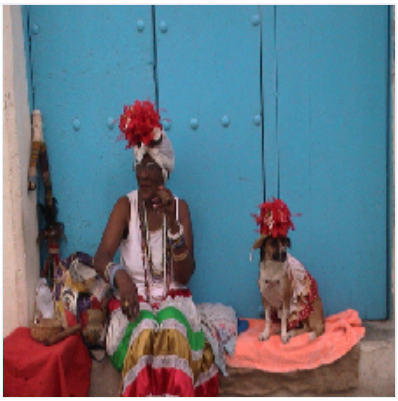
0.3864878714084625

0.3922061026096344

0.4144909679889679

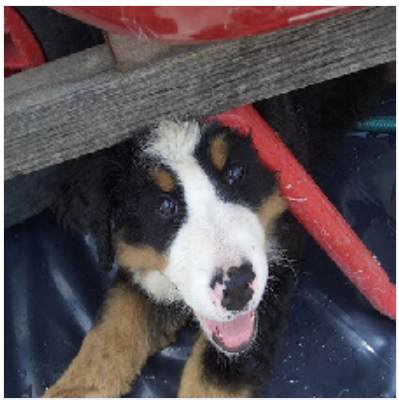


0.4158168137073517



0.4299579858779907

0.43139806389808655



0.4334023892879486

0.43694719672203064

0.43990498781204224

Now let's try with words we **did not train on**.

```
results = vector_search.search_index_by_value(word_vectors["ocean"], image_index, file_mappi
```

- [191, 'dataset/boat/2008_006720.jpg', 1.0978461503982544]
- [168, 'dataset/boat/2008_003951.jpg', 1.107574462890625]
- [156, 'dataset/boat/2008_001202.jpg', 1.1096670627593994]
- [192, 'dataset/boat/2008_006730.jpg', 1.1217926740646362]
- [198, 'dataset/boat/2008_007841.jpg', 1.1251723766326904]
- [158, 'dataset/boat/2008_001858.jpg', 1.1262227296829224]
- [184, 'dataset/boat/2008_005959.jpg', 1.1270556449890137]
- [173, 'dataset/boat/2008_004969.jpg', 1.1288203001022339]
- [170, 'dataset/boat/2008_004014.jpg', 1.1291345357894897]
- [180, 'dataset/boat/2008_005695.jpg', 1.129249930381775]

1.0978461503982544

1.107574462890625

1.1096670627593994

1.1217926740646362

1.1251723766326904

1.1262227296829224

1.1270556449890137

1.1288203001022339

1.1291345357894897

```
results = vector_search.search_index_by_value(word_vectors["tree"], image_index, file_mapping
```

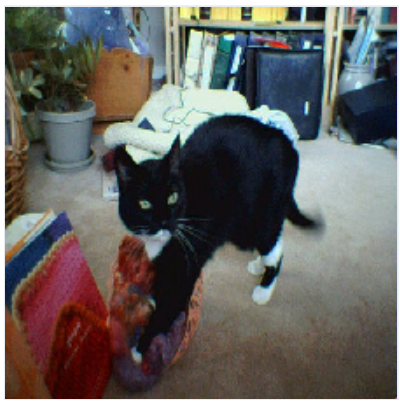
- [772, 'dataset/potted_plant/2008_005111.jpg', 1.085968017578125]
- [788, 'dataset/potted_plant/2008_007525.jpg', 1.090157389640808]
- [776, 'dataset/potted_plant/2008_005914.jpg', 1.0998691320419312]
- [775, 'dataset/potted_plant/2008_005874.jpg', 1.1043848991394043]
- [755, 'dataset/potted_plant/2008_001078.jpg', 1.104411005973816]
- [779, 'dataset/potted_plant/2008_006112.jpg', 1.1125234365463257]
- [774, 'dataset/potted_plant/2008_005680.jpg', 1.1161415576934814]
- [789, 'dataset/potted_plant/2008_007621.jpg', 1.1222350597381592]
- [405, 'dataset/chair/2008_001467.jpg', 1.1241838932037354]
- [502, 'dataset/dining_table/2008_000817.jpg', 1.126217246055603]

1.085968017578125

1.090157389640808

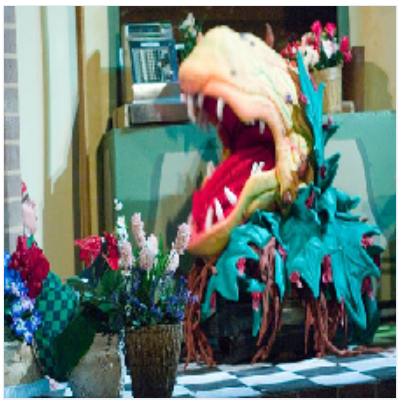
1.0998691320419312

1.1043848991394043



1.104411005973816

1.1125234365463257



1.1161415576934814

1.1222350597381592

1.1241838932037354

```
results = vector_search.search_index_by_value(word_vectors["street"], image_index, file_mapp
```

- [303, 'dataset/car/2008_000252.jpg', 1.1869385242462158]
- [579, 'dataset/dog/2008_004950.jpg', 1.2010694742202759]
- [72, 'dataset/bicycle/2008_005175.jpg', 1.201261281967163]
- [324, 'dataset/car/2008_004080.jpg', 1.2037224769592285]
- [284, 'dataset/bus/2008_007254.jpg', 1.2040880918502808]
- [741, 'dataset/person/2008_006554.jpg', 1.205135464668274]
- [273, 'dataset/bus/2008_005277.jpg', 1.2077884674072266]
- [274, 'dataset/bus/2008_005360.jpg', 1.211493730545044]
- [349, 'dataset/car/2008_008338.jpg', 1.2118890285491943]
- [260, 'dataset/bus/2008_003321.jpg', 1.214812159538269]

1.1869385242462158

1.2010694742202759

1.201261281967163

1.2037224769592285

1.2040880918502808

1.205135464668274

1.2077884674072266

1.211493730545044

1.2118890285491943

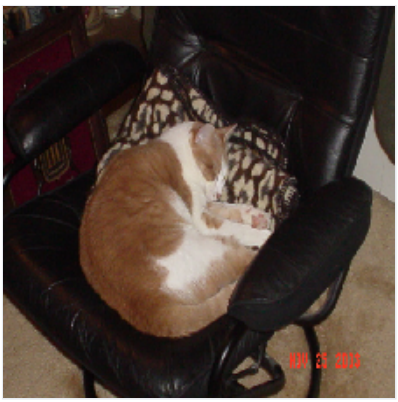
Our image search seems quite robust! Combining the semantics of images and words is quite powerful.

We can even search for a combination of multiple words by averaging word vectors together

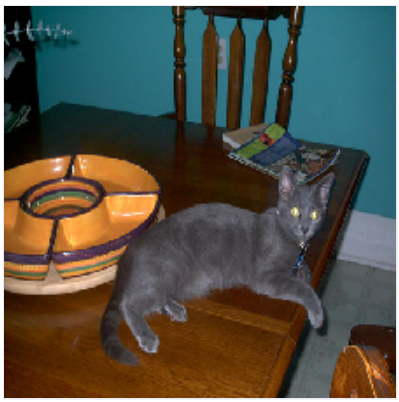
```
results = vector_search.search_index_by_value(np.mean([word_vectors["cat"], word_vectors["so
                                                image_index, file_mapping)
```

- [354, 'dataset/cat/2008_000670.jpg', 0.584483802318573]
- [395, 'dataset/cat/2008_007363.jpg', 0.6164626479148865]
- [352, 'dataset/cat/2008_000227.jpg', 0.6314529180526733]
- [891, 'dataset/sofa/2008_008313.jpg', 0.6323047876358032]
- [880, 'dataset/sofa/2008_006436.jpg', 0.6585681438446045]
- [369, 'dataset/cat/2008_003607.jpg', 0.7039458751678467]
- [386, 'dataset/cat/2008_006081.jpg', 0.7254294157028198]
- [885, 'dataset/sofa/2008_007169.jpg', 0.7283080816268921]
- [350, 'dataset/cat/2008_000116.jpg', 0.7311896085739136]
- [872, 'dataset/sofa/2008_004497.jpg', 0.7487371563911438]

0.584483802318573

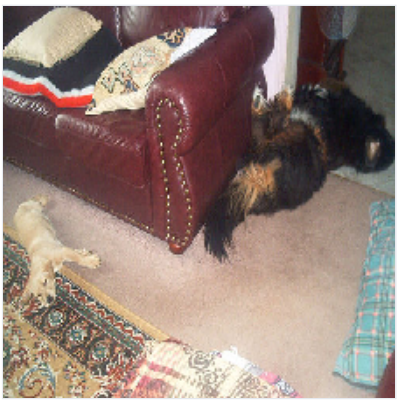


0.6164626479148865

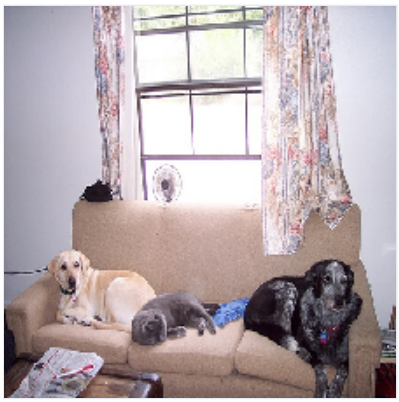


0.6314529180526733

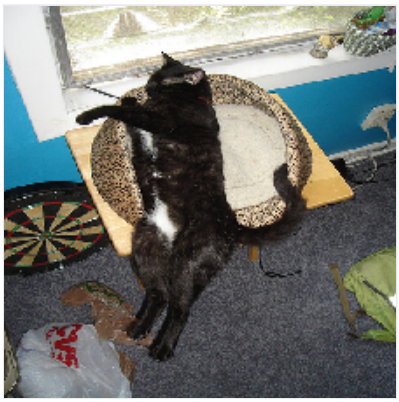
0.6323047876358032



0.6585681438446045



0.7039458751678467



0.7254294157028198

0.7283080816268921

0.7311896085739136

This is a fantastic result, as most those images contain some version of a furry animal and a sofa (I especially enjoy the leftmost image on the second row, which seems like a bag of furriness next to a couch)! Our model, which was only trained on single words, can handle combinations of two words. We have not built Google Image Search yet, but this is definitely impressive for a relatively simple architecture.

Hope you enjoyed it! Let me know if you have any questions, feedback, or comments.

Fin