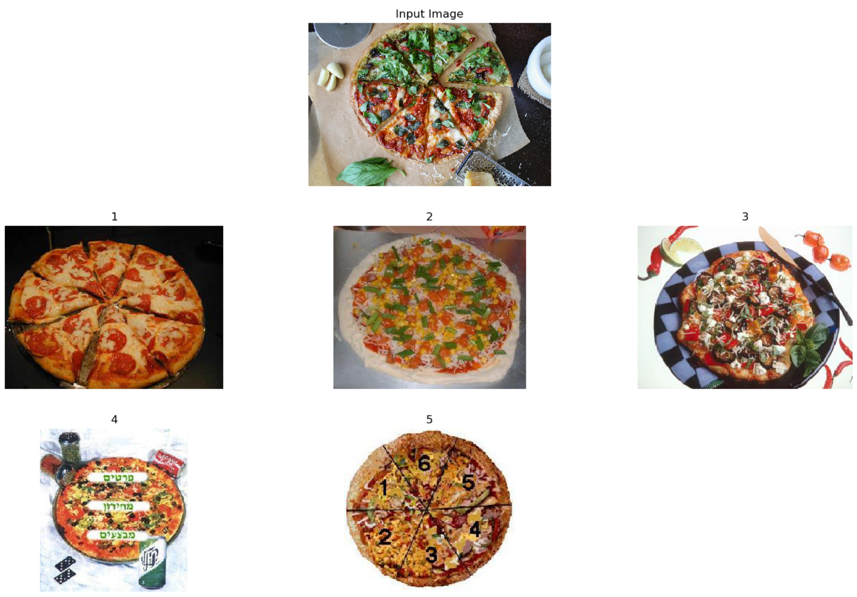
Finding similar images using Deep learning and Locality Sensitive Hashing

A simple walkthrough on finding similar images through image embedding by a ResNet 34 using FastAI & Pytorch. Also doing fast semantic similarity search in huge image embeddings collections.

Aayush Agrawal Follow
Mar 18 · 8 min read ★



Fina output with similar images given an Input image in Caltech 101

In this post, we are trying to achieve the above result, i.e., given an image, we should be able to find similar images from the Caltech-101 database. The post guides with an end to end process on how I went about building this. The entire codebase for replicating the project is in my GitHub repository. The process to achieve the above result can be broken down in these few steps -

1. Transfer learning from a ResNet-34 model(trained on ImageNet) to detect 101 classes in Caltech-101 dataset using FastAI and Pytorch.
2. Take the output of second last fully connected layer from trained ResNet 34 model to get embedding for all 9,144 Caltech-101

images.

3. Use Locality Sensitive hashing to create LSH hashing for our image embedding which enables fast approximate nearest neighbor search

4. Then given an image, we can convert it into image embedding using our trained model and then search similar images using Approximate nearest neighbor on Caltech-101 dataset.

Part 1—Data understanding and Transfer learning

As I mentioned above, for this project, my goal is to query any given image and find a semantically similar image in the Caltech-101 database. This database contains 9,144 images divided into 101 categories. Each category has about 50–800 images in them.

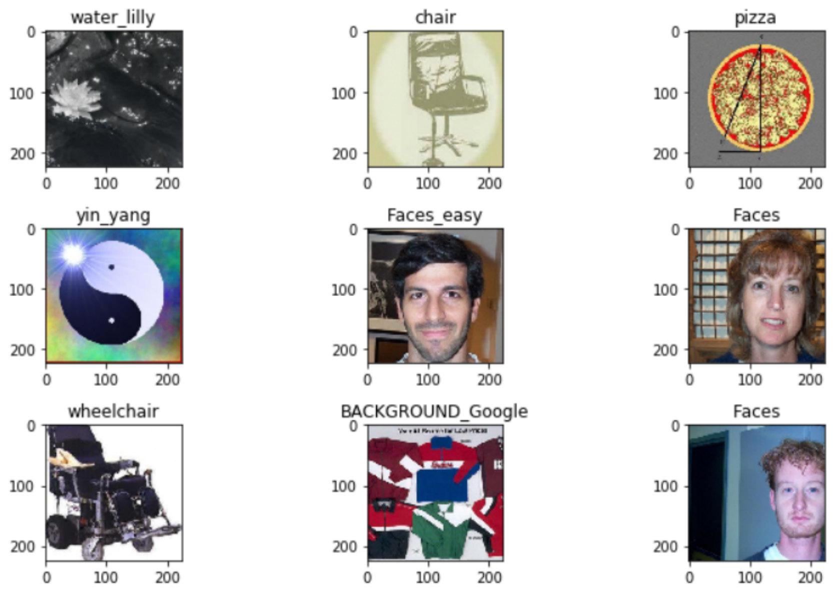


Image examples from Caltech-101 database

The first exercise in our project is to obtain a deep learning network which can classify these categories accurately. For this task, we will use a pre-trained ResNet 34 network which is trained on the ImageNet database and transfer learn it to classify 101 categories of Caltech-101 database using Pytorch 1.0 and FastAI library. As I have written about exactly how to do transfer learning with any given dataset in my previous blog, I am just going to outline the process in this blog. You can refer to this notebook to find the code to do the same. Find below the steps to do transfer learning for classifying Caltech-101 images -

1. Load the data using dataset loaders of Pytorch using FastAI library
2. Take a pre-trained network, in this case, a ResNet 34 and remove it's last fully connected layers
3. Add new fully connected layers at the end of the network and train only those layers using the Caltech-101 image, while keeping all the other layers frozen
4. Train the entire network by unfreezing all the layers

Part 2 — Extracting image embeddings using Pytorch Hooks

Now that we have a pre-trained network, we need to extract embeddings from this network for all of our Caltech-101 images. Embedding our nothing but a representation of an object in an N-dimensional vector. An image embedding, in this case, is a representation of an image in N-dimension. The basic idea is the closer a given image to another image their embedding will also be similar and close in the spatial dimension.

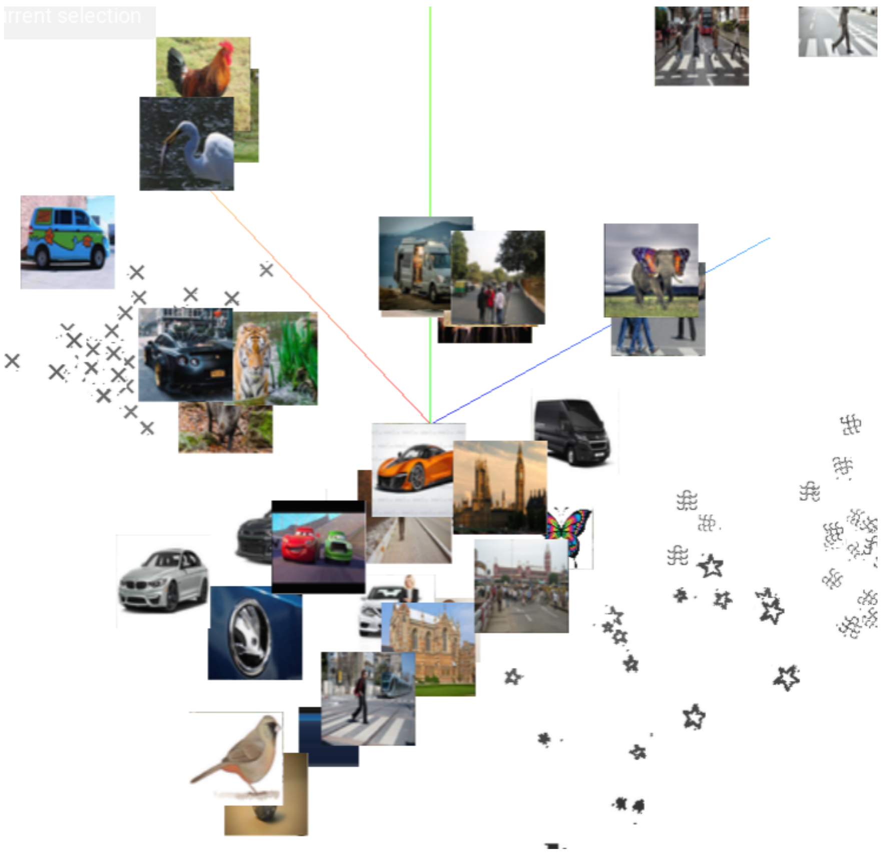


Image embedding visualization. Credit — Blog

You can see in the above image taken from this blog that image embedding is a spatial representation of an image in the vectorized form where similar images are close in spatial dimension as well.

We can obtain image embeddings from a ResNet-34 by taking the output of its second last Fully-connected layer which has a dimension of 512. To save intermediate calculations in a deep learning model in Pytorch for inspection or in our case to extract embeddings we use Pytorch Hooks. Hooks can be of two types—forward and backward. Forward hooks are used to save information passing forward in a network to make an inference while backward hooks are used to collect information about gradients during backpropagation. In our case, we need output of our second last Fully connected layers in the inference stage which means we need to use a forward hook. Let's look at the code for creating a hook (also in "Extracting Feature" section of my notebook) —

```
class SaveFeatures():
    features=None
    def __init__(self, m):
        self.hook = m.register_forward_hook(self.hook_fn)
        self.features = None
    def hook_fn(self, module, input, output):
        out = output.detach().cpu().numpy()
        if isinstance(self.features, type(None)):
            self.features = out
        else:
            self.features = np.row_stack((self.features, 
out))
    def remove(self):
        self.hook.remove()
```

The above code is all you need in creating a Pytorch hook. The SaveFeatures class invokes register_forward_hook function from the torch.nn module and given any model layer it will save the intermediate computation in a numpy array which can be retrieved using SaveFeatures.features functions. Let's see the code to use this class —

```
## Output before the last FC layer
sf = SaveFeatures(learn.model[1][5])
```

```
## By running this feature vectors would be saved in sf 
variable initated above
_= learn.get_preds(data.train_ds)
_= learn.get_preds(DatasetType.Valid)
```

```
## Converting in a dictionary of {img_path:featurevector}
img_path = [str(x) for x in
(list(data.train_ds.items)+list(data.valid_ds.items))]
feature_dict = dict(zip(img_path,sf.features))
```

Line 1–2: Invokes the class SaveFeatures using model layer reference to the output of second last fully-connected layer as the input.

Line 4–6: Passing the Caltech-101 data to get their predictions. Note that we are not interested in saving predictions and that's why we used "_." In this case, the intermediate output of second last layers in saved in the variable named "sf", which is an instance of SaveFeatures class.

Line 8–10: Creating a python dictionary where image path is the key and image embeddings is the value.

Now we have embedding representation of each image in Caltech-101 in our dictionary.

Part 3 — Locality Sensitive Hashing for fast approximate nearest neighbor search

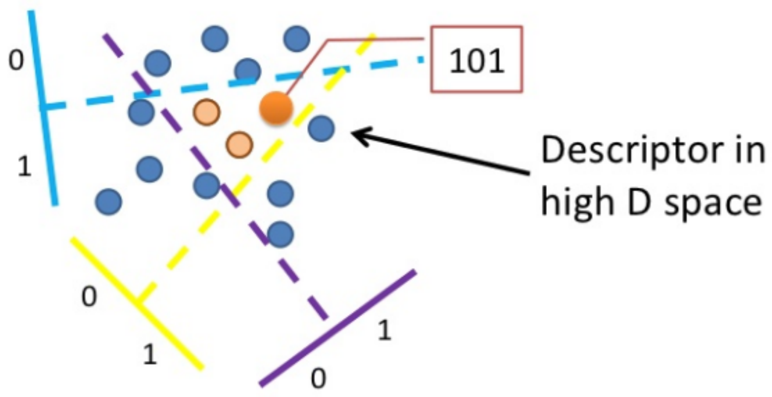
We can use our newly generated Caltech 101 image embeddings and get a new image, convert it into embedding to calculate distance b/w the new image and all the Caltech 101 database to find similar images. This process is computationally expensive in nature and as a new image embedding have to compare with all the 9K+ image embedding in the Caltech 101 database to find the most similar image(nearest neighbor), which in computational complexity notation is an $O(N^2)$ problem and will take exponentially more time to retrieve similar images as the number of images increases.

To solve this problem, we will use locality sensitive hashing(LSH) which is an approximate nearest neighbor algorithm which reduces the computational complexity to O(log N). This blog explains LSH in good details in terms of time complexity and implementation. In short, LSH generates a hash value for image embeddings while keeping spatiality of data in mind; in particular; data items that are similar in high-dimension will have a higher chance of receiving the same hash value.

Below are the steps on how LSH converts an embedding in a hash of size K-

1. Generate K random hyperplanes in the embedding dimension

2. Check if particular embedding is above or below the hyperplane and assign 1/0
3. Do step 2 for each K hyperplanes to arrive at the hash value



the hash value of the orange dot is 101 because it: 1) above the purple hyperplane; 2) below the blue hyperplane; 3) above the yellow hyperplane. Image Credit—Link

Let's now look at how LSH will perform an ANN query. Given a new image embedding, we will use LSH to create a hash for the given image and then compare the distance from image embedding of the pictures of Caltech-101 dataset which shares the same hash value. In this way, instead of doing similarity search over the whole Caltech-101 database we will only do a similarity search with a subset of images which shares the same hash value with the input image. For our project, we are using lshash3 package for an approximate nearest neighbor search. Let's look at the code to do the same (you can find the code in the "Using Locality Sensitive hashing to find near similar images" section of my notebook)-

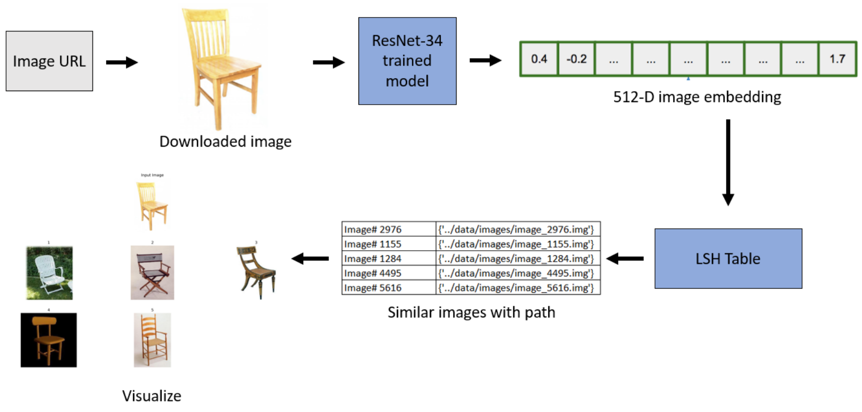
```
from lshash import LSHash

k = 10 # hash size
L = 5  # number of tables
d = 512 # Dimension of Feature vector
lsh = LSHash(hash_size=k, input_dim=d, num_hashtables=L)

# LSH on all the images
for img_path, vec in tqdm_notebook(feature_dict.items()):
    lsh.index(vec.flatten(), extra_data=img_path)
```

The above code takes the image embedding dictionary and converts it into LSH table. To query the LSH table, we can use the code below —

```
# query a vector q_vec
response = lsh.query(q_vec, num_results= 5)
```

Part 4—Putting it all together

Now we have our LSH table created let's write a script which can take an image URL as an input and give us N(user-defined) similar images from CalTech 101 database. The code for this part is on my Github here.



Image# 2976	{'../data/images/image_2976.img'}
Image# 1155	{'../data/images/image_1155.img'}
Image# 1284	{'../data/images/image_1284.img'}
Image# 4495	{'../data/images/image_4495.img'}
Image# 5616	{'../data/images/image_5616.img'}

Process flow of the find similar image script.

The script does the following task -

1. Load the LSH table and our ResNet 34 model (load_model function)
2. Take the image URL from user call and download the image (download_img_from_url function)
3. Pass the image from ResNet-34 to get 512 dimension image embedding (image_to_vec function)
4. Query it with LSH table to find N(user-defined) similar images and their path (get_similar_images function)
5. Return the output at the desired output path, optionally display it using Open CV (get_similar_images function)

We can use a similar concept in various applications like finding similar images in our photo gallery, item-item recommendation of similar looking items, doing a web search on images, finding near-duplicate images, etc.

Summary (TL;DR).

In the blog, we saw an application of deep learning in finding *semantically similar images* and how to do an *approximate nearest neighbor* query using Locality-sensitive hashing(LSH) to speed up query time for large datasets. Also, it's important to note that we used LSH not on the raw features(images) but on the embeddings which help do *fast similarity search in huge collections.*

I hope you enjoyed reading, and feel free to use my code on Github to try it out for your purposes. Also, if there is any feedback on code or just the blog post, feel free to reach out on LinkedIn or email me at aayushmnit@gmail.com. You can also follow me on Medium and Github for future blog post and exploration project codes I might write.