from scratch

The unreasonable effectiveness of Deep Learning Representations

Emmanuel Ameisen Follow
Jul 5, 2018 · 16 min read

Teaching computers to look at pictures the way we do

Want to learn applied Artificial Intelligence from top professionals in Silicon Valley or New York? Learn more about the Artificial Intelligence program at Insight.

Are you a company working in AI and would like to get involved in the Insight AI Fellows Program? *Feel free to get in touch.*

. . .

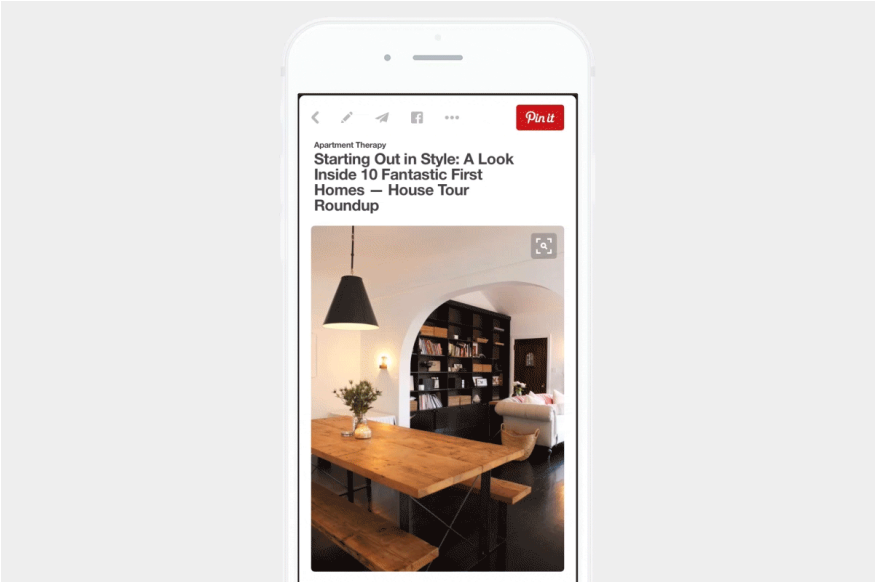
For more content like this, follow Insight and Emmanuel on Twitter.

Why similarity search?

An image is worth a thousand words, and even more lines of code.

Many products **fundamentally appeal to our perception**. When browsing through outfits on clothing sites, looking for a vacation rental on Airbnb, or choosing a pet to adopt, the way something looks is often an important factor in our decision. The way we perceive things is a strong predictor of what kind of items we will like, and therefore a valuable quality to measure.

However, making computers understand images the way humans do has been a computer science challenge for quite some time. Since 2012, Deep Learning has slowly started overtaking classical methods such as Histograms of Oriented Gradients (HOG) in perception tasks like image classification or object detection. One of the main reasons often credited for this shift is deep learning's ability to automatically **extract meaningful representations** when trained on a large enough dataset.



Visual search at Pinterest

This is why many teams—like at Pinterest, StitchFix, and Flickr—started using Deep Learning to learn representations of their images, and **provide recommendations** based on the content users find visually pleasing. Similarly, Fellows at Insight have used deep learning to build models for applications such as helping people find cats to adopt, recommending sunglasses to buy, and searching for art styles.

Many recommendation systems are based on collaborative filtering: leveraging user correlations to make recommendations ("users that liked the items you have liked have also liked…"). However, these models require a **significant amount of data** to be accurate, and struggle to handle **new items** that have not yet been viewed by anyone. Item representation can be used in what's called content-based recommendation systems, which do not suffer from the problem above.

In addition, these representations allow consumers to **efficiently search** photo libraries for images that are similar to the selfie they just took (querying by image), or for photos of particular items such as cars (querying by text). Common examples of this include Google Reverse Image Search, as well as Google Image Search.

Based on our experience providing technical mentorship for many semantic understanding projects, we wanted to write a tutorial on how you would go about **building your own representations**, both for image and text data, and **efficiently do similarity search**. By the end of this post, you should be able to build a quick semantic search model from scratch, no matter the size of your dataset.

This post is accompanied by an annotated code notebook using streamlit and a self-standing codebase demonstrating and applying all these techniques. Feel free to run the code and follow along!

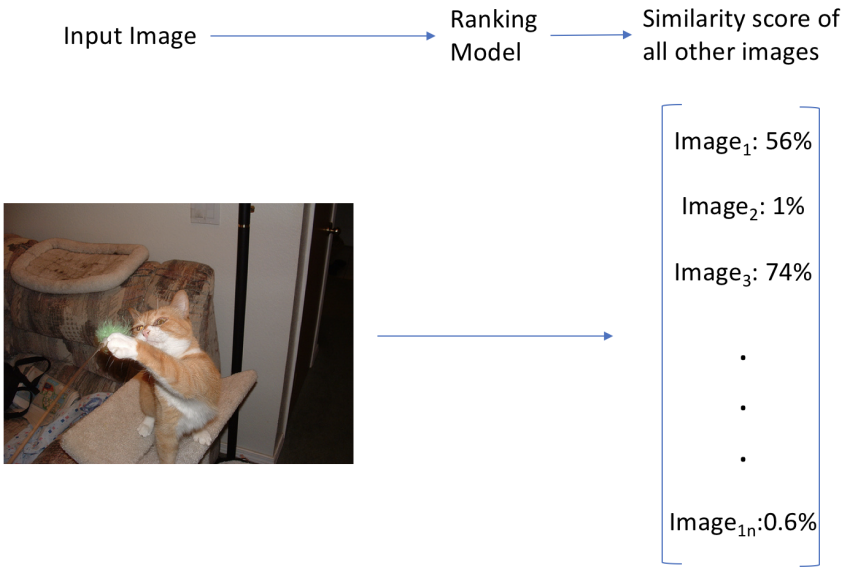
. . .

What's our plan?

A break to chat about optimization

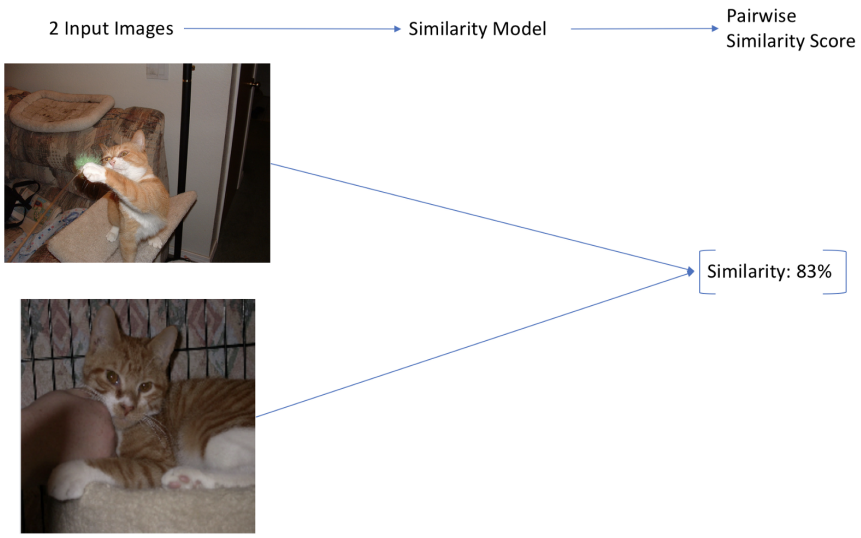
In machine learning, just like in software engineering, there are many ways to tackle a problem, each with different tradeoffs. If we are doing research or local prototyping, we can get away with very inefficient solutions. But if we are building an image similarity search engine that needs to be maintainable and scalable, we have to consider both how we can **adapt to data evolution**, and **how fast our model can run**.

Let's imagine a few approaches:



Workflow for approach 1

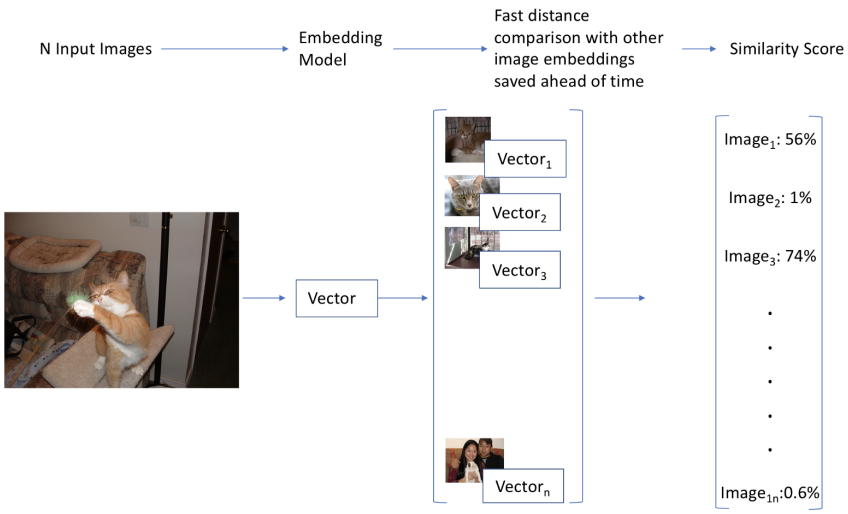
1/ We build an end-to-end model that is trained on all our images to take an image as an input, and output a similarity score over all of our images. Predictions happen quickly (one forward pass), but we would need to **train a new model** every time we add a new image. We would also quickly reach a state with so many classes that it would be extremely **hard to optimize** it correctly. This approach is fast, but does not scale to large datasets. In addition, we would have to label our dataset by hand with image similarities, which could be extremely time consuming.



Workflow for approach 2

2/ Another approach is to build a model that takes in two images, and outputs a pairwise similarity score between 0 and 1 (Siamese

Networks[PDF], for example). These models are accurate for large datasets, but lead to another scalability issue. We usually want to find a similar image by **looking through a vast collection of images**, so we have to run our similarity model once for each image pair in our dataset. If our model is a CNN, and we have more than a dozen images, this becomes too slow to even be considered. In addition, this only works for image similarity, not text search. This method scales to large datasets, but is slow.



Workflow for approach 3

3/ There is a simpler method, which is similar to word embeddings. If we find an **expressive vector representation, or embedding** for images, we can then calculate their similarity by looking at **how close their vectors are to each other**. This type of search is a common problem that is well studied, and many libraries implement fast solutions (we will use Annoy here). In addition, if we calculate these vectors for all images in our database ahead of time, this approach is both fast (one forward pass, and an efficient similarity search), and scalable. Finally, if we manage to find **common embeddings** for our images and our words, we could use them to do text to image search!

Because of its simplicity and efficiency, the third method will be the focus of this post.

How do we get there?

So, how do we actually use **deep learning representations** to create a **search engine?**

Our final goal is to have a search engine that can take in images and output either similar images or tags, and take in text and output similar

words, or images. To get there, we will go through three successive steps:

- Searching for **similar images to an input image** (Image → Image)
- Searching for **similar words to an input word** (Text → Text)
- Generating **tags for images**, and **searching images using tex**t (Image ↔ Text)

To do this, we will use **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 find these is by calculating the cosine similarity between our image embedding, and embeddings for other images. Similar images will have similar embeddings, meaning a **high cosine similarity between embeddings**.

Let's start with a dataset to experiment with.

Dataset

Images

Our image dataset consists of a total of a **1000 images**, divided in 20 classes with 50 images for each. This dataset can be found here. Feel free to use the script in the linked code to automatically download all image files. *Credit to Cyrus Rashtchian, Peter Young, Micah Hodosh, and Julia Hockenmaier for the dataset.*

This dataset contains a category and a set of captions for every image. To make this problem harder, and to show how well our approach generalizes, we will **only use the categories**, and disregard the captions. We have a total of 20 classes, which I've listed out below:

`aeroplane` `bicycle` `bird` `boat` `bottle` `bus` `car` `cat` `chair` `cow` `dining_table` `dog` `horse` `motorbike` `person` `potted_plant` `sheep` `sofa` `train` `tv_monitor`

Image examples. As we can see, the labels are quite noisy.

We can see our labels are pretty **noisy**: many photos contain multiple categories, and the label is not always from the most prominent one. For example, on the bottom right, the image is labeled `chair` and not `person` even though 3 people stand in the center of the image, and the chair is barely visible.

Text

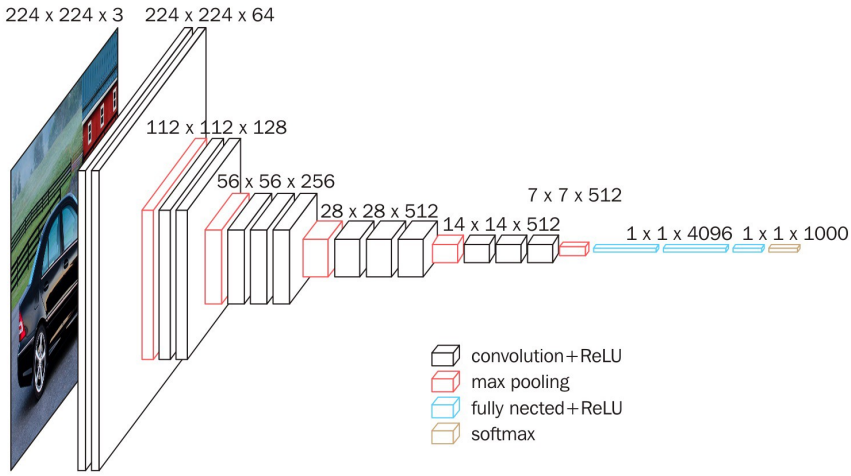
In addition, we load word embeddings that have been pre-trained on Wikipedia (this tutorial will use the ones from the GloVe model). We will use these vectors to incorporate text to our semantic search. For more information on how these word vectors work, see Step 7 of our NLP tutorial.

Image -> Image

Starting simple

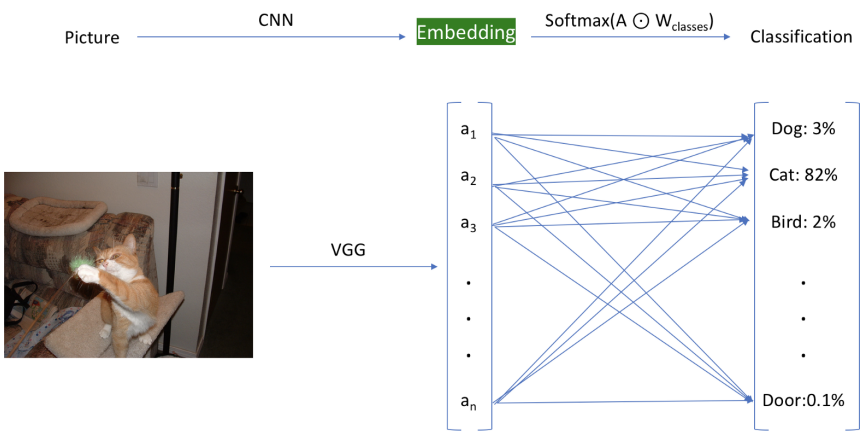
We are now going to load a model that was **pre-trained** on a large data set (Imagenet), and is freely available online. We use VGG16 here, but

this approach would work with any recent CNN architecture. We use this model to generate **embeddings** for our images.



VGG16 (credit to Data Wow Blog)

What do we mean by generating embeddings? We will use our pre-trained model **up to the penultimate layer**, and store the value of the activations. In the image below, this is represented by the embedding layer highlighted in green, which is before the final classification layer.



For our embeddings, we use the layer before the final classification layer.

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 reasons that embeddings are so popular in practical applications, as they allow for huge efficiency gains. On top of storing them to disk, we will build a **fast index** of the embeddings using Annoy, which will allow us to very quickly find the nearest embeddings to any given embedding.

Below are our embeddings. Each image is now represented by a sparse vector of size 4096. *Note: the reason the vector is sparse is that we have taken the values after the activation function, which zeros out negatives.*

	:4	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095
0	0	0.4546	0	6.1516	0	1.3763	0	0	1.1756	0.4681	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1.8355	0	0	0	0	0.6923	0	0	0
3	:5	0	0	0	0.4420	5.2478	0	0	1.6778	0	0	0
4	:6	0.1924	0	0	0	0	0	0	0.8986	0	0	0
5	:9	0	0	2.0729	0	0	0	0	0.0996	0.2377	0	0
6	0	0	0	6.8556	0	1.3900	0	0	0.6859	1.1272	0	0
7	0	0	0	2.2498	0	0	0	1.8188	0.1840	0	0	0
8	0	0	0	6.0193	0	0	0	0	0	0	0	0
9	:5	0	0	0	0.9562	0	0.3197	0	1.8738	3.5308	0	1.3911
10	:6	0	0	0.2099	0	0	0	1.4480	1.3150	1.1056	2.1684	0

Image Embeddings

Using our embeddings to search through images

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

This is especially useful since image labels are often **noisy**, and there is more to an image than its label.

For example, in our dataset, we have both a class `cat`, and a class `bottle`.

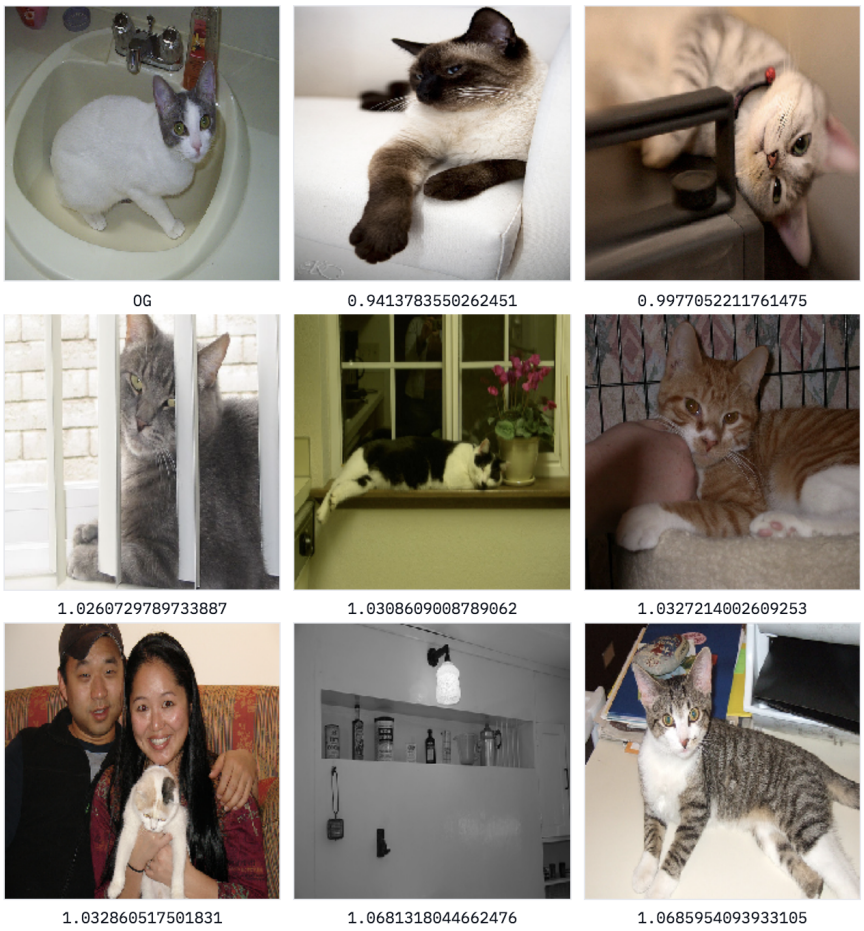
Which class do you think this image is labeled as?

Cat or Bottle? (Image rescaled to 224x224, which is what the neural network sees.)

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 similar images to `dataset/bottle/2008_000112.jpg` …



Great—we mostly get more images of **cats**, which seems very reasonable! Our pre-trained network has been trained on a wide variety of images, including cats, and so it is able to accurately find similar images, even though it has never been trained on this particular dataset before.

However, one image in the middle of the bottom row shows a shelf of bottles. This approach perform wells to find similar images, in general, but sometimes we are only interested in **part of the image**.

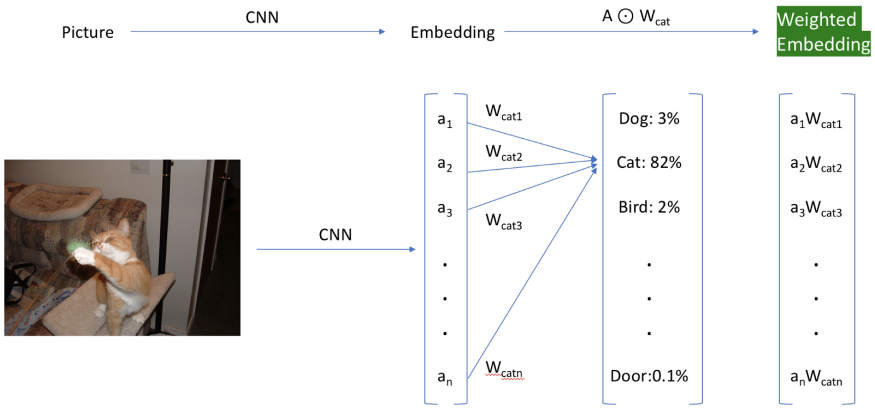
For example, given an image of a cat and a bottle, we might be only interested in similar cats, not similar bottles.

Semi-supervised search

A common approach to solve this issue 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. We do this by loading the last layer of weights we had initially discarded, and only use the weights tied to the index of the class we are looking for to re-weigh our embedding. This cool trick was initially brought to my attention by Insight Fellow Daweon Ryu. For example, in the image below, we use the weights of the `Siamese cat` class to re-weigh the activations on our dataset (highlighted in green). Feel free to check out the attached notebook to see the implementation details.



The hack to get weighted embeddings. The classification layer is shown for reference only.

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

Searching for similar images to `dataset/bottle/2008_000112.jpg` using weighted features…

We can see that the search has been biased to look for **Siamese cat-like things**. We no longer show any bottles, which is great. You might however notice that our last image is of a sheep! This is very interesting, as biasing our model has led to a **different kind of error**, which is more appropriate for our current domain.

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 1000 **Imagenet classes**. These classes are far from all-encompassing (they lack a category for people, for example), so we would ideally like to find something more **flexible.** In addition, what if we simply wanted to search for cats **without providing an input image?**

In order to do this, we are going to use more than simple tricks, and leverage a model that can understand the semantic power of words.

Text -> Text

Not so different after all

Embeddings for text

Taking a detour to the world of natural language processing (NLP), we can use a similar approach to **index and search for words**.

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

Just like before, we will create an index, this time containing all of the GloVe vectors. Then, we can **search our embeddings** for similar words.

Searching for `said`, for example, returns this list of [`word`, `distance`]:

- `['said', 0.0]`
- `['told', 0.688713550567627]`
- `['spokesman', 0.7859575152397156]`
- `['asked', 0.872875452041626]`
- `['noting', 0.9151610732078552]`
- `['warned', 0.915908694267273]`
- `['referring', 0.9276227951049805]`
- `['reporters', 0.9325974583625793]`
- `['stressed', 0.9445104002952576]`
- `['tuesday', 0.9446316957473755]`

This seems very reasonable, most words are quite similar in meaning to our original word, or represent an appropriate concept. The last result (`tuesday`) also shows that this model is far from perfect, but it will get us started. Now, let's try to incorporate both words and images in our model.

A sizable issue

Using the distance between embeddings as a method for search is a pretty general method, but our representations for words and images seem **incompatible**. The embeddings for images are of size 4096, while those 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 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**.

Image <-> Text

Worlds collide

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

For the first time in this tutorial, we will actually be training our own model, drawing inspiration from a great paper called DeViSE. We will not be re-implementing it exactly, though we will heavily lean on its main ideas. (For another slightly different take on the paper, check out fast.ai's implementation in their lesson 11.)

The idea is to combine both representations by re-training our image model and **changing the type of its labels**.

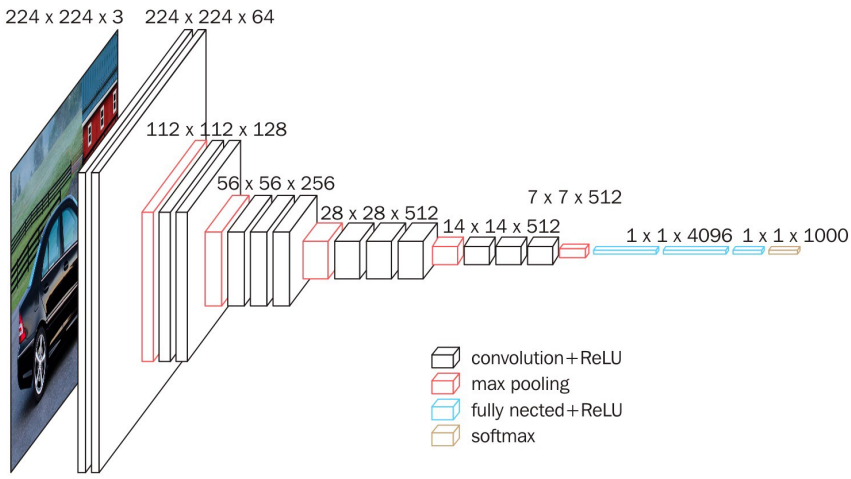
Usually, image classifiers are trained to pick a category out of many (1000 for Imagenet). What this translates to is that—using the example of Imagenet—the last layer is a vector of size 1000 representing the **probability of each class**. This means our model has no semantic understanding of which classes are similar to others: classifying an image of a `cat` as a `dog` results in as much of an error as classifying it as an `airplane`.

For our hybrid model, we replace this last layer 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, and means that similar classes will be closer to each other (as the word vector for `cat` is closer to `dog` than `airplane`). Instead of a target of size 1000, with all zeros except for a one, we will predict a **semantically rich word vector** of size 300.

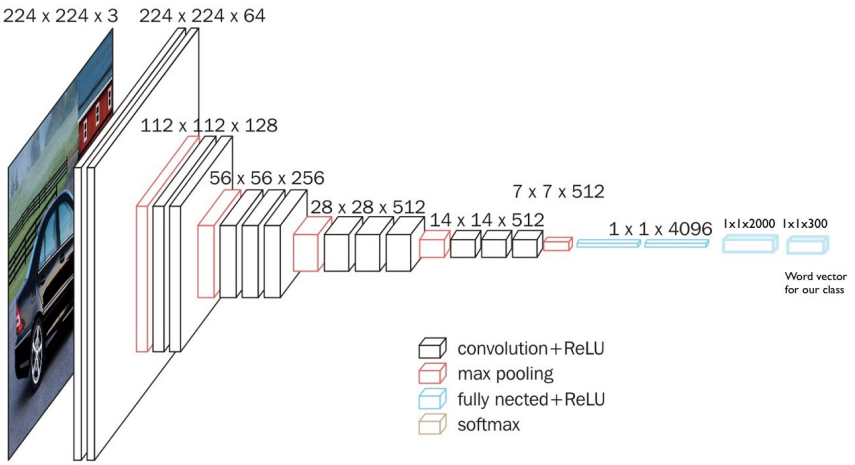
We do this by adding two dense layers:

- One intermediate layer of size 2000
- One output layer of size 300 (the size of GloVe's word vectors).

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



Here is what it looks like now:



Training the 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 the 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, but is still much faster than on Imagenet. For reference, it took about 6–7 hours on my laptop that has **no GPU**.

It is important to note how ambitious this approach is. 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 very well on the test set, and would certainly not expect it to perform at all on completely new examples.

Once our model is trained, we have our GloVe word index from above, and we build a new fast index of our image features by running all images in our dataset through it, saving it to disk.

Tagging

We can now easily extract tags from any image by simply feeding our image to our trained network, saving the vector of size 300 that comes out of it, and finding the closest words in our index of English words from GloVe. Let's try with this image—it's in the `bottle` class, though it contains a variety of items.

Here are the generated tags:

- `[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]`

That's a pretty amazing result, as most of the tags are **very relevant**. This method still has room to grow, but it picks up on most of the items in the image fairly well. 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 (which we get by running them through our model).

Generalized image search with minimal data.
Let's start first with a word that was actually in our training set by searching for `dog:`

Results for search term "dog"

OK, pretty good results—but we could get this from any classifier that was trained just on the labels! Let's try something harder by searching for the keyword `ocean`, which is not in our dataset.

Results for search term "ocean"

That's awesome—our model understands that `ocean` is similar to `water`, and returns many items from the `boat` class.

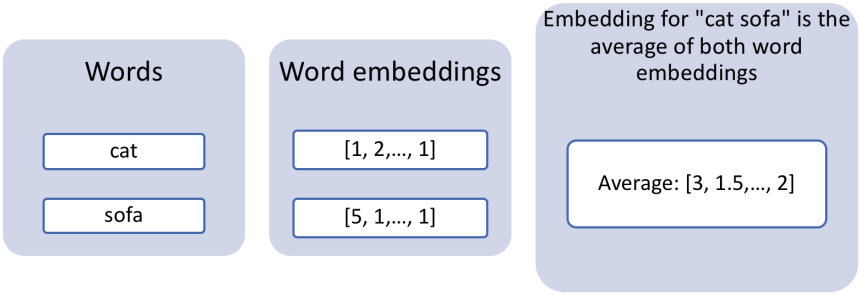
What about searching for `street`?

Results for search term "street"

Here, our images returned come from plenty of classes (`car` , `dog` , `bicycles` , `bus` , `person`), yet most of them contain or are near a street, despite us having never used that concept when training our model. Because we are **leveraging outside knowledge** through pre-trained word vectors to learn a mapping from images to vectors that is more **semantically rich** than a simple category, our model can generalize well to outside concepts.

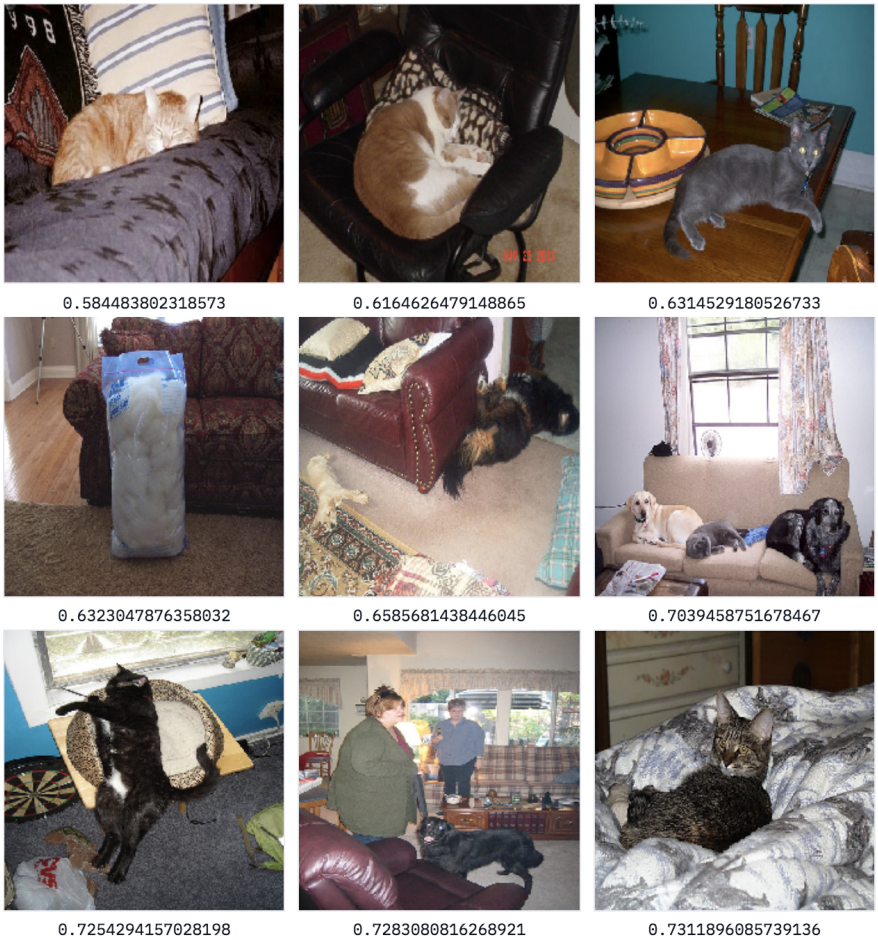
Beyond words

The English language has come far, but not far enough to invent a word for everything. For example, at the time of publishing this article, there is no English word for "a cat lying on a sofa," which is a perfectly valid query to type into a search engine. If we want to search for multiple words at the same time, we can use a very simple approach, leveraging the arithmetic properties of word vectors. It turns out that summing two word vectors generally works very well. So if we were to just search for our images by using the average word vector for `cat` and `sofa` , we could hope to get images that are very cat-like, very sofa-like, or have a cat on a sofa.



Getting a combined embedding for multiple words

Let's try using this hybrid embedding and searching!



Results for search term "cat"+"sofa"

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.

This method can actually extend quite naturally to a variety of domains (see this example for a joint code-English embedding), so we'd love to hear about what you end up applying it to.

Conclusion

I hope you found this post informative, and that it has demystified some of the world of content-based recommendation and semantic search. If you have any questions or comments, or want to share what you built using this tutorial, reach out to me on Twitter!

. . .

Want to learn applied Artificial Intelligence from top professionals in Silicon Valley or New York? Learn more about the Artificial Intelligence program.

Are you a company working in AI and would like to get involved in the Insight AI Fellows Program? *Feel free to get in touch.*