Lenny #2: Autoencoders and Word Embeddings

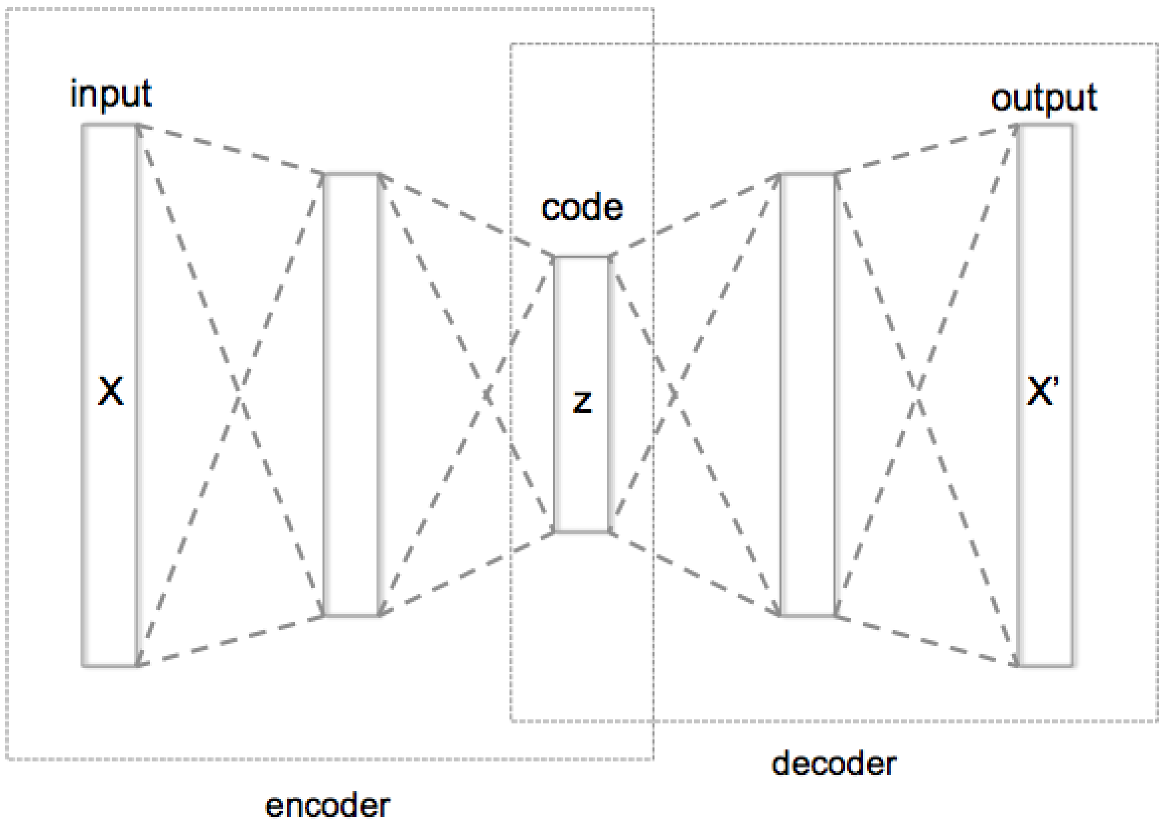
Lenny Khazan
Apr 24, 2016 · 12 min read

I want to take a quick break from my reinforcement learning endeavor (more on that soon) to talk about an interesting unsupervised learning model: autoencoders. The basic idea behind autoencoders is **dimensionality reduction** — I have some high-dimensional representation of data, and I simply want to represent the same data with fewer numbers. Let's take the example of a picture of a face. If you look at each individual pixel, you'll quickly realize that neighboring pixels are usually highly correlated (they have a very similar color). There's a lot of redundant information going on there. What if we could take out the redundancy and express that same image in a fraction of the numbers?

That's where autoencoders come in.

Autoencoders

Remember how neural networks work? Autoencoders are a kind of neural network designed for dimensionality reduction; in other words, representing the same information with fewer numbers. The basic premise is simple — we take a neural network and train it to spit out the same information it's given. By doing so, we ensure that the activations of each layer **must**, by definition, be able to represent the entirety of the input data (if it is to be successfully recovered later on). If each layer is the same size as the input, this becomes trivial, and the data can simply be copied over from layer to layer to the output. But if we start changing the size of the layers, the network inherently learns a new way to represent the data. If the size of one of the hidden layers is smaller than the input data, it has no choice but to find some way to compress the data.



https://en.wikipedia.org/wiki/File:Autoencoder_structure.png

And that's exactly what an autoencoder does. If you look at the diagram, the network starts out by "compressing" the data into a lower-dimensional representation z, and then converts it back to a reconstruction of the original input. If the network converges properly, z will be a more compressed version of the data that encodes the same information. For example, if the input image is a face, the data can be reduced down to certain defining characteristics of a face — shape, color, pose, and so on. Representing each of those characteristics could be more effective than storing each pixel, and that's what autoencoders are really great at.

It's often helpful to think about the network as an "encoder" and a "decoder". The first half of the network, the encoder, takes the input and transforms it into the lower-dimensional representation. The decoder then takes that lower-dimensional representation and converts it back to the original image (or as close to it as it can get). The encoder and decoder are still trained together, but once we have the weights, we can use the two separately — maybe we use the encoder to generate a more meaningful representation of some data we're feeding into another neural network, or the decoder to let the network generate new data we haven't shown it before.

That’s pretty much it for the basic concept of autoencoders, but there are a few modifications that people have made that are worth mentioning.

Denoising Autoencoders

Denoising autoencoders are a technique often used to help the network learn representations of the data that are more meaningful to the underlying data’s variability. Instead of simply training a network to recall the input it was given, random noise is applied to the input before passing it to the network — the network is still expected to recall the original (uncorrupted) input, which should force the network to stop picking up on minute details while focusing on the bigger picture. The random noise essentially prevents the network from learning the specifics, ensuring that it generalizes to the important characteristics. This is especially important in the situation where the encoded vector is larger than the input — using a traditional autoencoder, the network could (and often will) simply learn to copy the input into the encoded vector to recreate the original. With a denoising autoencoder, the autoencoder can no longer do that, and it’s more likely to learn a meaningful representation of the input.

Sequence-to-sequence Autoencoders

We haven’t covered recurrent neural networks (RNNs) directly (yet), but they’ve certainly been cropping up more and more — and sure enough, they’ve been applied to autoencoders. The basic premise behind an RNN is that, instead of operating on and producing a fixed-size vector, we instead start operating on and producing **sequences** of fixed-size vectors. Let’s take the example of machine translation (here’s a sentence in English, give me the same sentence in Russian). Posing this problem to a vanilla neural network is intractable (we could translate the sentence word-for-word, but we already know grammar doesn’t work like that). Enter RNNs: we give the network the input sentence one word at a time, and the network magically spits out the same sentence in Spanish, one word at a time. We’ll get more into the specifics of how exactly an RNN works soon enough, but the general concept is enough for the time being.

Sequence-to-sequence autoencoders work the same way as traditional autoencoders, except both the encoder and decoder are RNNs. So instead of converting a vector to a smaller vector, we convert an entire sequence into one vector. Then we take that one vector and expand it back into a sequence. Let’s take the example of getting a fixed-length representation of a variable-length audio clip. The autoencoder takes in an audio segment and produces a vector to represent that data. That's the encoder. We can then take that vector and give it to the decoder, which gives back the original audio clip.

Variational Autoencoders

I won't go into depth on how variational autoencoders work now (they really deserve a post of their own — maybe this summer), but the concept is still important enough that it's worth mentioning. If you're curious (and comfortable with some heavy-duty statistics), this paper is worth a read.

Variational autoencoders essentially let us create a **generative model** for our data. A generative model learns to create brand new data that isn't from the training set, but that looks as if it is. For example, given a dataset of faces, it will learn to generate brand new faces that look real. Variational autoencoders aren't the only generative model — generative adversarial networks are another kind of generative model that we'll take a look at some other day. (Forgive me for the handy-wavy explanation, but specifics are hard without going into the statistics. Rest assured I'll give variational autoencoders the explanation they deserve soon enough.)

Word Embeddings

Now I want to talk about a different (but somewhat related) type of model known as a **word embedding**. Like an autoencoder, this type of model learns a **vector space embedding** for some data. I kind of skipped over this point earlier on, so let me take a minute to address this.

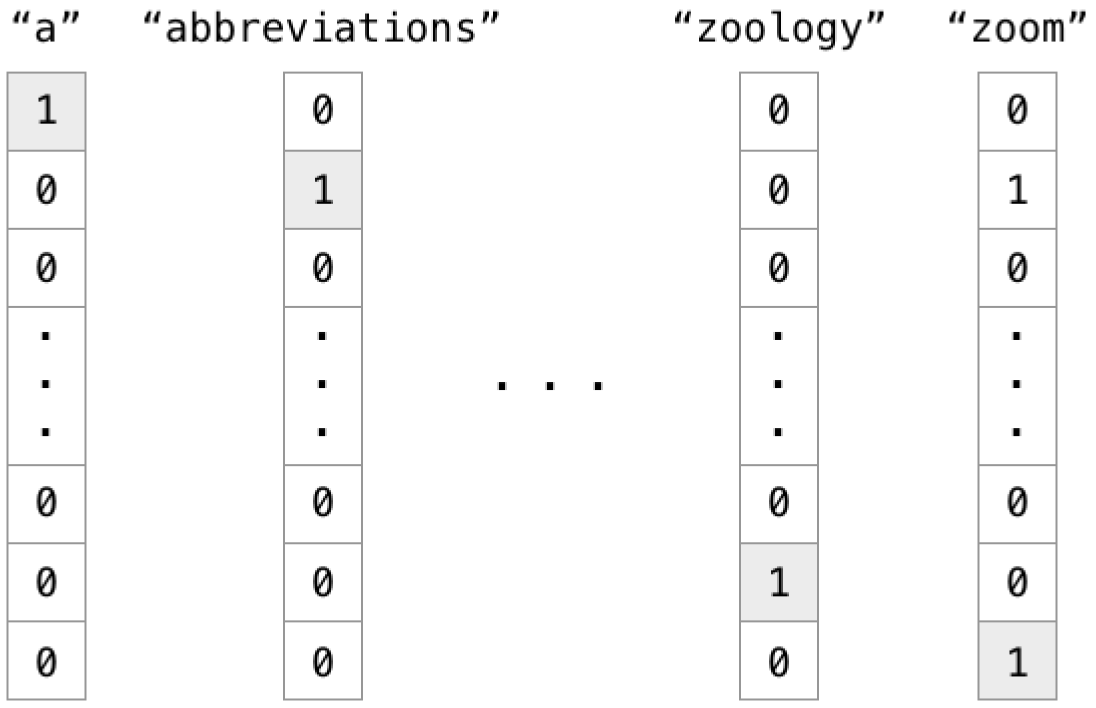
Let's go back to the example of faces one last time. We started out by representing faces as a series of pixel values (let's say 10,000 pixels per face) — for simplicity's sake, we'll assume that the image is grayscale and each pixel is represented by a single intensity value, but know that the same principles carry over for RGB data (and, of course, non-image data too). Now, let's do something a bit interesting, and maybe unnatural at first, with these pixels. Let's use each pixel value as a coordinate in a 10,000-dimensional space. (Sidenote: if you're having trouble visualizing this, it's because we mortal humans can't think in more than 3 dimensions — just think of everything in terms of 2 or 3 dimensions, but realize that the exact same principles transfer over to as many dimensions as you want). So we have a 10,000-dimensional space, and and the 10,000 pixel values are the coordinates into this space. Each image in our dataset has a distinct location in this space. If you were to look at this impossible-to-visualize vector space, you'll notice that pictures that have very similar pixel values are very close to each other, while very different images are very far away from each other.

This relationship between images is critical, and it's what makes it useful to think about data in vector form. But there are flaws in the coordinate system we're using now — for example, if you have an identical image but shifted over one pixel, it could be nowhere near the original image in this vector space because each dimension will have a wildly different coordinate. If we can find some coordinate system that gives us more meaningful relationships between points in our vector space, we can put our images into a space that tells us even more about the relationships between these images.

And if you think about it, that's exactly what neural networks do! We take an image, which is represented by very low-level pixel data, and convert it into a higher-level representation that might encode features like pose or facial expression. If we train our neural network on, for example, a classification task, we'll get a new representation that best encompasses the differences between the various classes. If we're using an autoencoder, the new representation will instead tell us more about the factors that best define what is unique about a particular face. If we use these new encoded vectors as coordinates into a new coordinate system, we'll see that the relationship between images suddenly becomes even more meaningful — images that are close together are similar not because of pixel-by-pixel similarity, but because they have similar characteristics. Instead of having a dimension per pixel, we have a dimension for every feature of the face (pose, color, etc.). As a result, as we move along a certain axis, we notice that an individual feature of the image changes while the rest remains the same (for example, we might notice that a smile gradually becomes a frown, while skin color and pose remain the same). If we were to try the same thing on our original pixel-based coordinate system, we'll just see the intensity of one individual pixel change, which is much less meaningful.

So this process of taking a point in one vector space (the pixel-based one) and putting it into a different vector space (the autoencoder-based one) is a vector space embedding. We are embedding the points of one vector space into another, which could tell us something new about our data that we didn't quite realize before.

The process of creating a meaningful vector space embedding is usually *relatively* straightforward for something like faces. If two faces have similar appearance, they should appear next each other in the vector space. Traditional autoencoders are pretty good at this. But the relationships aren't always quite so obvious — let's look at the example of words. If I look at the letters that make up a word, it's darn near impossible for me to make sense of what words should be near each other. I might notice that "presentation" and "present" have similar letters, so they should be next to each other; but on the other hand, "prequel" and "sequel" also have similar letters, yet they have a very different relationship. Similarly, "yacht" and "boat" are quite related, but don't have much in common letter-wise. As a result, words are usually encoded in vector form using **one-of-k encoding** (also known as **one-hot encoding**). If we have a total of N possible words, then each vector will have N numbers, and all but one of the numbers will be a 0. Each word will have its own index into the vector, and the value at that position will be a 1. While this does let us differentiate between words, it gives us no information about how words are related or linked.



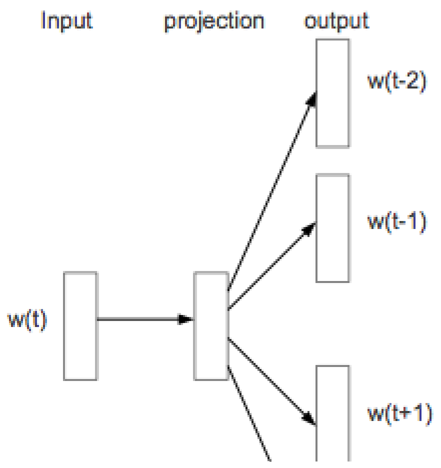
This is what one-hot encoding looks like. It's convenient, but gives no information about the relationship between words.

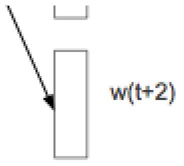
We need a better system. Instead of using one-hot encoding, what if we can use something like an autoencoder to reduce our words to a lower-dimensional vector that gives us more information about the meaning and characteristics of a word itself? Sounds like a plan! But, as we already established, autoencoders can't learn much meaning from the letters that make up a word. We'll need to set up a new model to learn the meanings of and relationships between words.

And that is exactly what a **word embedding** is. We'll take a look at a popular word embedding model shortly, but first let's explore some of the fascinating properties a word embedding vector space can have. As it turns out, and this shouldn't come as much of a surprise, words have much more complicated relationships than pictures of faces. For example, take "Washington D.C." and "United States". The two are obviously related, but the relationship is very specific. It's also the same relationship that "Ottawa" and "Canada" have. And both of those relationships are very different than the relationship between "boy" and "girl". Ideally, a word embedding model will be able to express each of these relationships distinctly.

When you have a vector space, you can express relationships as the vector between two points. So, if you take the distance from "boy" to "girl", traveling the same distance (in the same direction) should get you from "man" to "woman", because they two have the same relationship. But the direction you take to get from "Ottawa" to "Canada" should be completely different. If we can create a word embedding space that captures each of these relationships, that means that we'll now have an incredibly useful and meaningful way to represent words. Once we can get words into a space like this, neural networks (and of course other ML models) will be able to learn much more effectively — while neural networks would typically have to inherently learn a simplified version of these relationships on their own, the ability to give them this knowledge directly means that it's easier to perform natural language processing tasks (like, for example, machine translation).

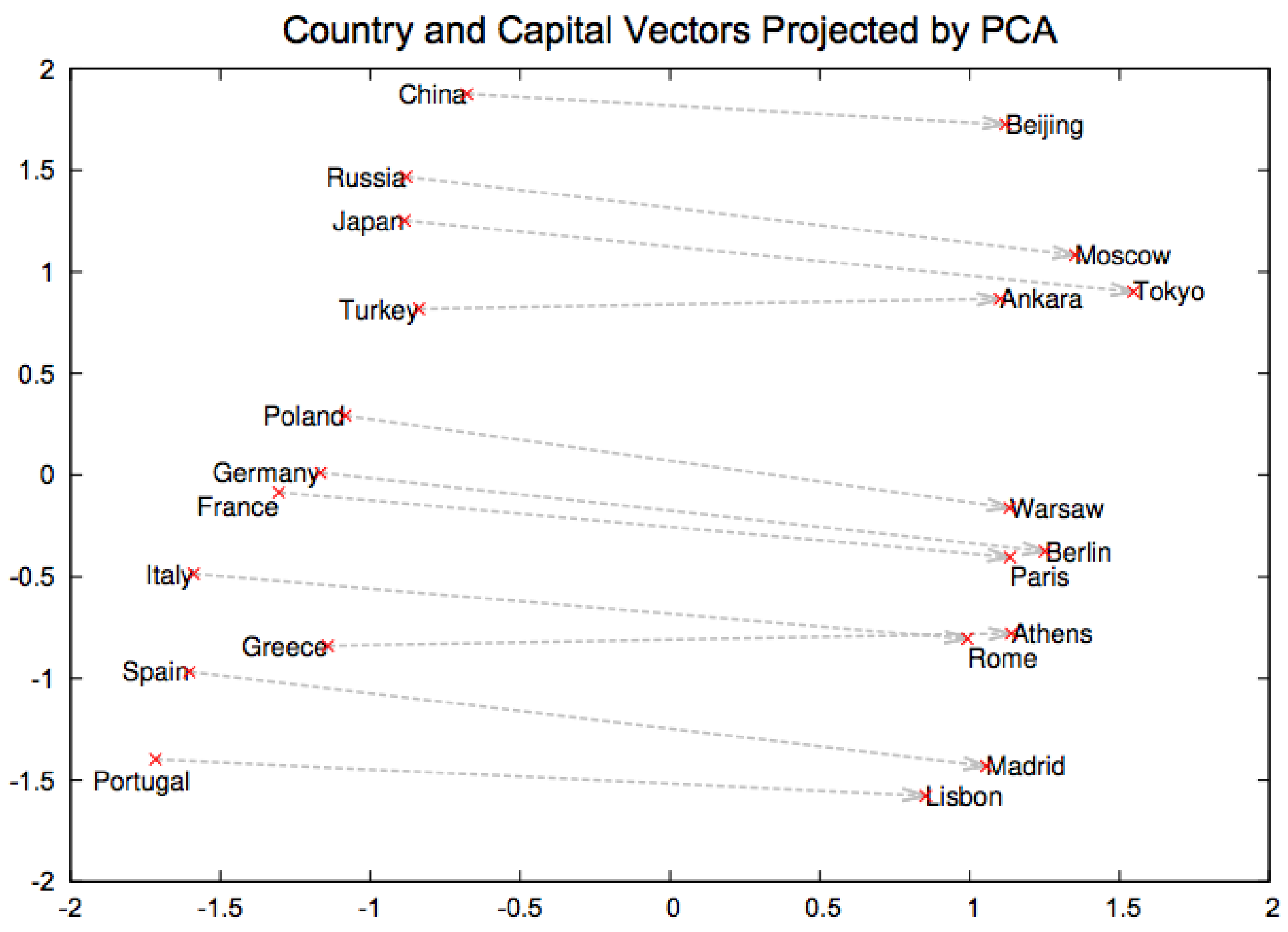
Now, finally, we can talk about how we get these word embeddings. I'm going to talk about one particularly impressive model called **word2vec**. At a very high level, it learns its word embedding through context. This actually makes lots of sense — when you see a word you don't know, you look at the words around it to figure out what it means. As it turns out, word2vec does the exact same thing.





https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

Generally, word2vec is trained using something called a **skip-gram model**. The skip-gram model, pictures above, attempts to use the vector representation that it learns to predict the words that appear around a given word in the corpus. Essentially, it uses the context of the word as it is used in a variety of books and other literature to derive a meaningful set of numbers. If the “context” of two words is similar, they will have similar vector representations — that already makes this new embedding more useful than one-hot encoding. But, as it turns out, the relationships go deeper.



https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf. This is a visualization of the relationships between countries and their capitals — in reality, the vector space has hundreds of dimensions, but a technique called PCA was used to approximate this relationship in a way that’s easier to visualize. As you can see, the relationship is very similar in each instance. If you want to see some more incredible results, definitely check out the word2vec paper linked above.

If you look at the relationship between a country and its capital, you'll notice that the vector taking you from one to the other is almost identical in every instance. That is one of the other critical characteristics we defined earlier for a word embedding — representing the unique relationships that words have with each other.

Let's look at a neural network trained for machine translation to get a better idea of how this new word embedding helps us. Giving it words as one-hot vectors is not super useful — it has to learn on its own that the vector for "screen" and "display" have the same meaning, even though there is no correlation whatsoever between the two vectors. But, using our fancy new word2vec embedding, "screen" and "display" should have a nearly identical vector representations — so this correlation comes very naturally. In fact, even if the network has never seen a training sample with the word "display" in it, it will likely be able to generalize based purely on the examples it *has* seen with "screen" because the two have a similar representation.

So yeah, autoencoders are pretty neat.

There's lots more I didn't get to cover about autoencoders (most notably pre-training comes to mind, but that's for another day), but hopefully you learned enough to understand where they might come in handy and a few ways they can be used in practice. We also talked about embeddings and word2vec, a word embedding model that gives us meaningful word vectors to make our other models more effective. But the utility of word2vec extends far beyond NLP — just last year, a word2vec-based system of creating an embedding space for protein sequences was proposed for improved accuracy on classification tasks.

The applications for these unsupervised learning models are practically endless, and they generally serve to improve the accuracy of more traditional supervised learning problems. That being said, I'm looking forward to seeing what else comes out of the field in the coming years — the most exciting advances are still to come.

Algorithms Machine Learning