8 Word Embeddings towards LLMs

The ability of computer programs to effectively process, categorize, and generate language has been a recent and momentous advancement in AI. We have so far seen the k-gram approaches, which revolutionized how we could categorize and search document-level objects – which propelled web search in the early 2000s. However, these next level challenges need to represent language as a more granule level – so each word (or potentially subword) is given an abstract, easy-to-use representation. Namely, a high-dimensional vector.

How do we represent words as abstract objects? The classic approach to natural language processing was through linguistics, and breaking sentences down into semantic structures of verbs, nouns, prepositions, conjunctions, etc, and their use as subjects, objects, actions, etc. While this provided useful context for analysis, it was hard – and did not fit naturally with the rest of developing data mining and machine learning infrastructure.

The key to unlocking language was, (1) mapping words to high-dimensional vectors where we could use Euclidean distance and cosine similarity, and (2) embracing a simplistic ansatz:

The Distributional Hypothesis: *Words that occur in the same contexts have similar meanings.*
(c.f., Zellig Harris, J.R. Firth)

8.0.1 Word Vectors

This approach is data driven; it takes a very large corpus of text (say all of Wikipedia – as a *small* example) and generates a vector representation for each word. And in this setting, the Euclidean distance or cosine similarity is most appropriate. This is useful for many more detailed natural language applications. For starters synonyms typically have vectors close to each other, and in general all verbs are closer to each other than all nouns, or all adjectives. At a finer level, these embeddings usually reveal more structure like there is a direction that captures properties like gender (from man to woman, etc). Moreover, one can sometimes solve analogy questions. The most famous example is to take the representative vectors for king (v_{king}), queen (v_{queen}), and man (v_{man}), and then consider the vector manipulation

$$\hat{v} = v_{\mathsf{king}} - v_{\mathsf{queen}} + v_{\mathsf{man}}.$$

Impressively, the nearest vector to $\hat{v}$ is often v_{woman}, the vector representation of woman.

8.1 Three Generations of Word Vector Embedded Representations

These methods basically follow 3 generations:

8.1.1 PPMI Vectors from Aggregated CBOW

A common preliminary approach, starts with an understandable, but very high-dimensional, vector called a PPMI (positive pointwise mutual information) vector for each word. This is extremely high-dimensional, often too high to easily work with or represent explicitly.

PPMI vectors. We start with defining a fixed vocabulary we care about; lets say the $m = 100{,}000$ most common distinct words. Then we define a window around each word in the text corpus. The window may be the 3 words on either side of the work in question; or more recently, this *context window* is often only the $k \gg 3$ words preceding the word in question. Then for each word, lets say $i = \mathsf{octopus}$, we keep a count of how often it appears n_{octopus} and the number of times each other vocabulary word co-occurs with it within

a window (i.e., continuous bag of words). For word i, define its number of cooccurrences with word j as $b_{i,j}$. Let these counts be a vector $b_i \in \mathbb{R}^m$ for word i with the jth coordinate as $b_{i,j}$.

Now define the probability of a word i as $p(i) = n_i/N$ (where there are a total of N words in the corpus we consider), and the co-occurence probability as $p(i,j) = b_{i,j}/N$. Then the pointwise mutual information of i and j as $\log \frac{p(i,j)}{p(i)p(j)}$. Ultimately we create a vector $v_i \in \mathbb{R}^m$ for each word, in which we enforce each coordinate to be non-negative. The jth coordinate is then the PPMI between i and j, defined as

$$v_{i,j} = \max\left(0, \log \frac{p(i,j)}{p(i)p(j)}\right).$$

Now for the ith word, this leads to a vector:

$$v_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,j}, \ldots, v_{i,m}) \in \mathbb{R}^m.$$

As a whole v_i captures a representation of the co-occurence patters of the ith word with all of the other words. Connecting back to our distributional hypothesis, when words i and i' are similar if $\mathbf{d}_{\mathsf{Euc}}(v_i, v_{i'}) = \|v_i - v_{i'}\|$ is small or $\mathbf{d}_{\mathsf{cos}}(v_i, v_{i'})$ is small; that is they have similar word distributions in their aggregated context windows.

8.1.2 Self-supervised Embeddings

Traditional supervised machine learning takes as input a data set $\{(x_1, y_1), (x_2, y_2), \ldots\}$ where each $x_i \in \mathcal{X}$ (typically $\mathcal{X} = \mathbb{R}^d$) and each y_i is an associated label (often $y_i \in \{-1, +1\}$). It is classical to think of each label being witnessed from a past observational study, or assigned by an expert (e.g., doctor assesses cancer / no-cancer). Supervised machine learning is very powerful to identify patterns in the $X = \{x_i\}_i$ which correlate with the labels in y_i ... as long as there is enough data for the models to generalize.

In the last 20 years, we started to be able to generate, store, and compute-on much larger data sets X, but our ability to get expert labels has mostly not kept up. To build larger and more general models we sought to engineer more labels.

Self-supervision is the process where a machine learning model is set up to try to predict the input data (or part of it), itself.

An early example were auto-encoders for images, where a large image (1000s of pixels large) is passed through a two functions (often as a neural network): an *Encoder* to a lower-dimensional space, and then from there to a *Decoder* which attempts to recover the original pixels. So the input and desired output are the same! In this case, the reason the function cannot just be the identity is that the intermediate *latent* layer is much lower-dimensional than the input size, so it is forced to somehow capture some compact (and hopefully meaningful representation of the data). Unless the data family has a true low-dimensional latent space, this inevitably loses some information – perhaps one can argue it "denoises" but in practice, it tends to add noise to the output. That is, auto-encoded reconstructed images often appear blurry than the originals.

In text, another option is available: a **masked model**. In this setting, we do not try to predict the entire input, but only a subset of each input x_i. In images, this could be masking out a square of the image that is hidden from the input, and predicted by the output. In text, this aligns perfectly with the CBOW formulation, as we mask a single word from a string of text (the context window).

Around 2013-14 the Word2Vec (Google) and GloVe (Stanford) models were introduced using CBOW and masked models to build self-supervised learning for text. However, the training did not just predict the masked word, but it did so via an embedding. It used a neural network (LSTMs) where the last layer was a moderate dimensional vector (e.g. $d = 300$ dimensions). Each word (of say $m = 100{,}000$) was assigned a vector $v_{\mathsf{word}} \in \mathbb{R}^d$ and the prediction was based on the nearest word (under cosine distance). The network was trained for this high-dimensional nearest-neighbor task.

The resulting output was an embedding in $\mathbb{R}^d$ for each word (among say $m = 100{,}000$) in the English language. This quickly became the default starting point for most natural language processing tasks, and several long-standing benchmarks for tasks within that field quickly saw improvement comparable to a decade of progress.

Moreover, from these representations arose emergent behaviors. Word types tended to group together. Linear subspaces (e.g., defining male-female gender) were clearly observable. Analogies could be solved by vector transport (e.g., $v_{\mathsf{France}} + (v_{\mathsf{London}} - v_{\mathsf{Paris}}) \approx v_{\mathsf{England}}$).

While the $100{,}000$-dimensional PPMI vectors probably would have worked similarly, these worked very well, and were several of orders of magnitude smaller, which made things much more efficient to work with.

So what was the self-supervised learning, learning? The co-occurrence properties for the English language. In the case where the context window was **before** the predicted word, it was learning in a string of text, what was likely to come next: *next token prediction*.

What should we make of un-intended correlations? What if the male-female subspace was aligned with the engineer-secretary subspace? That this occurs, may not be surprising given the sort of text (all of the internet) that these models are trained on. But should we use this for all prediction tasks involving input language?

Generalized mask-model embeddings. These ideas from self-supervised masked model embeddings has wide use beyond language. More famously, DeepWalk and Node2Vec masked nodes in random walks in graphs to embed the nodes of a graph in a vector space. Learning masked parts of images can give image-patch embeddings. Similar approaches have led to embeddings for spatial data, geometry objects, genomic data, financial transactions etc. The resulting useful representations are sometimes called "foundational models."

8.1.3 Contextual Transformer Embeddings

This was followed by the 3rd generation (roughly where we are now), although there are many nuances that are specialized for language, and beyond the scope of this class. This was driven by two advances that came almost on top of each other.

Contextual embeddings. These gave each instance of a word its own vector representation, based on its specific context. This was addressing well-known concern that homonyms (words with two or more distinct meanings – like apple the company and apple the fruit) would be embedded in the same vector location. These words and their representations did not work well in linguistic tasks, somehow splitting the difference between the multiple ideal positions of their distinct meanings. Moreover, even within a single meaning, there were nuances lost.

Instead this learned a function $f_\theta : [\mathsf{context}{+}\mathsf{word}] \to \mathbb{R}^d$. The parameters θ of the function were trained using self-supervised learning. This was pioneered as ELMo (Allen Institute for AI & U Washington) where f used a bidirectional LSTM (type of neural networks for sequences).

Transformer with attention. ELMo ran into issues where it could only use a limited context size, since sometimes far away words (at start of sentence, or 3 sentences earlier) was very influential in the words meaning. Moreover negation (in its many subtle forms) was still a real challenge to deal with. This was addressed with a recent break through of transformer neural architectures with *attention*. This would take a long sequence of token (e.g., words in context) and determine which were mostly like to influence the prediction at hand.

BERT and then RoBERTa (Google) were the prototypes for this method, and are still roughly near the state-of-the-art. These were the models that were adapted by GPT at OpenAI and were an essential part of their ChatGPT chat bot and subsequent models.

As ELMo and RoBERTa are neural networks, they operates in layers. And while traditionally the last layer can be seen as the embedding layer, the previous layers can be useful as well for various prediction tasks. The first layer roughly acts as a non-contextual network (like GloVe, Word2Vec), and the subsequent layers (to a first approximation) gradually add more context to the representation.

Another change these models made was using *subwords* instead of words. These included most very common words, but for uncommon words, it would break them into their basewords (e.g., latin or greek bases), and could generalize better to rarely seen words. The details of how this was done is along the lines of k-gram modeling, and involves modeling and many engineered choices.