Reading Assignment 3: Embeddings

If I'm pretraining from scratch for peak monolingual English, I prefer to learn token embeddings end-to-end inside the model and let them co-adapt with the tokenizer, the loss, and the attention stack, because that's what captures polysemy and domain nuance once the higher layers contextualize the tokens. The pros are straightforward: end-to-end learning aligns the geometry of the embedding space with the rest of the network, lets me tie weights to the LM head, and plays nicely with subword or byte tokenizers so I never see out-of-vocabulary artifacts. The cons are mostly engineering: the vocabulary size times the model width can dominate parameters and VRAM, and a too-large learning rate on the embedding table can inflate norms early or overfit to frequent tokens. I mitigate that by using no weight decay on the embedding table, applying a slightly smaller learning rate to embeddings than to the transformer blocks—think on the order of thirty to seventy percent of the backbone rate—and adding a touch of dropout around five to ten percent at the embedding output; those three knobs keep optimization smooth without giving up capacity.

By contrast, the pre-defined embeddings in the readings, like word2vec, are static, word-level vectors that don't change with context. Their advantage is that they can warm-start cheaply when data is tiny, they're easy to inspect, and they can stabilize early training if I had to operate with very limited corpora. In a from-scratch LLM, though, they become a mismatch: word-level granularity doesn't line up with subword or byte tokenization, they lock me into a geometry the transformer didn't learn with, and they ignore polysemy entirely, so they tend to cap perplexity and downstream quality. If I were forced to use them, the only thing I'd do is use them to initialize a few rows for high-frequency English words as a gentle prior and then immediately resume full end-to-end training with a low learning rate on those seeded rows so the space can adapt to the transformer; in practice, with real pretraining data, I'd skip this altogether for monolingual English.

On positional information, the sinusoidal encoding from the readings is a solid historical baseline because it's parameter-free and gives me smooth distance signals that extrapolate modestly beyond the trained length, so for a short window around four thousand tokens it's perfectly serviceable and often within striking distance of stronger choices. Learned absolute positional embeddings can be strong inside the exact range I trained, but they don't extrapolate, so if I ever need to push past the original max length I'm stuck doing positional interpolation and a short continued pretrain to recover performance, and even then quality drops off as I stretch far beyond the trained context. For both short and long windows I prefer relative or rotary schemes; with monolingual English at four

thousand tokens, rotary embeddings are typically the best default because they cost essentially no extra parameters and tend to outperform sinusoids slightly, and with one-hundred-twenty-eight-thousand tokens, rotary with NTK or xPos-style scaling or ALiBi biases gives me robust distance awareness and graceful length generalization that absolute tables just don't provide. A quiet but important detail here is to treat sequence length as a curriculum variable: even if I set a 128k window, if I don't feed many near-max-length sequences during training, the model will learn "short," so I deliberately mix in a large fraction of very long samples and keep some short ones for optimization efficiency.

Zooming back to the token embeddings under these two window regimes, for English at four thousand tokens I'll learn embeddings inside the model, tie them to the LM head, train a BPE or Unigram tokenizer with byte fallback in roughly the fifty-thousand-token range so I never see true OOVs, and run a conventional optimization schedule while monitoring length-sliced perplexity and embedding-norm histograms. For English at one-hundred-twenty-eight-thousand tokens I do the same but I'm more careful about table size; if the product of vocabulary and width starts creeping past about a hundred million parameters for the table alone, I factorize the embeddings by learning them in a smaller subspace and projecting up to the model width, accept that I won't be able to tie weights exactly in that case, and compensate with a slightly longer warmup and conservative gradient clipping. In the long-window setting I also lower the embedding learning rate relative to the backbone and keep weight decay off any position-bias parameters, because these components are the first places I see norm drift when I scale length.

Putting the readings into perspective, the sine positional encoding and word2vec represent two ends of a "fixed versus learned" axis. Fixed sinusoids are lightweight and general but leave some performance on the table; fixed word embeddings are easy to inspect but fundamentally misaligned with a modern LLM's subword pipeline and contextualization. Learning both token embeddings and positional structure inside the model tends to dominate for peak English quality because it lets the network discover the geometry that best serves its own attention patterns, and replacing the fixed sinusoid with rotary or ALiBi preserves the good parts of relative distance while adding the long-range stability I need if I ever push beyond four thousand tokens. The main tradeoff is compute and memory in the embedding table, which I would handle with weight tying when possible and factorization when necessary, and a modest increase in training care for long windows where length curriculum and scaled positional frequencies become critical.