Lecture 24: Language Models and Transformers

CS 3780/5780, Sp26 – Sarah Dean

April 28, 2026

The basic idea of today's large language models (which power chatbots and agents) is surprisingly simple:

> meaning need not be understood, only predicted.

The consequence of this idea is that supervised machine learning is sufficient for building language models. Before we get into how that works, we will start with the origins of this idea.

Part 1: Prediction, Generation, and Claude Shannon

Claude Shannon was an electrical engineer and mathematician who spent WWII at Bell Labs working on cryptography and secure communications. These were problems that forced him to think carefully about the structure of information. At home, Claude Shannon and his wife Mary played the following game. Claude would pick up a book and turn to a page at random. Mary would guess letters one at a time; when she was wrong, Claude would supply the correct letter to keep things going. Like hangman, the goal was to require as few provided letters as possible.

Figure 1: Claude and Mary Shannon at home.

```
(1) THE ROOM WAS NOT VERY LIGHT A SMALL OBLONG
(2) ----ROO------NOT-V-----I------SM----OBL----
(1) READING LAMP ON THE DESK SHED GLOW ON
(2) REA----------O------D----SHED-GLO--O--
(1) POLISHED WOOD BUT LESS ON THE SHABBY RED CARPET
(2) P-L-S------O--BU--L-S--O------SH-----RE--C-----
```

Letters represent hints given by Claude, dashes are Mary's correct guesses. From Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1), 50–64.

These prediction games are possible because there are *statistical regularities* in language. One obvious regularity is letter frequency: in English, e is far more common than z. Richer regularities exist: after q, the next letter is almost always u, after *th* the next letter is likely e, and after *the*, the next word is very likely a noun or adjective. Patterns compound, so the more context you have, the better your prediction. Mary would often rattle off long correct streaks by completing "reading lamp" and "polished wood".

Shannon formalized this in his famous 1948 report *"A Mathematical Theory of Communication"* by thinking about *orders of approximation* to English. Each order uses more context:

The zero-th order samples letters uniformly at random.

```
XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZLHJQD
```

The 1st order samples letters according to their frequency in English.

```
OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL
```

The 2nd order samples each letter conditioned on the previous one (bigram statistics).

```
ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT
TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE.
```

The 3rd order samples each letter conditioned on the previous one (trigram statistics).

```
IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME OF DEMONSTURES OF THE
REPTAGIN IS REGOACTIONA OF CRE.
```

The Word-level 1st order samples words according to their frequency.

```
REPRESENTING AND SPEEDILY IS AN GOOD APT OR COME CAN DIFFERENT NATURAL HERE HE
THE A IN CAME THE TO OF TO EXPERT GRAY COME TO FURNISHES THE LINE MESSAGE HAD BE THESE.
```

The Word-level 2nd order samples each word conditioned on the previous one.

```
THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE
ANOTHER METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER TOLD THE PROBLEM FOR AN UNEXPECTED.
```

Notice that generating these samples and predicting the next token are two sides of the same coin. To generate 2nd-order letter text, Shannon needed to know, for each letter he'd written so far, the likelihood of each coming next. That is exactly a prediction model. Conversely, any model which predicts probabilities can be turned into a generator: sample a token from the predicted distribution, append it, and repeat. The statistical regularities Mary was exploiting to guess letters are the same regularities Shannon used to generate sentences.

Furthermore, each order of approximation is a better model of language, and each step produces more recognizable text. Notice the jump in realism when we consider words rather than letters. What is the right atomic unit of prediction? Before we can build a prediction model, we need to decide what we are predicting over. The atomic units are called **tokens** t, and the set of all token types is the **vocabulary** $\mathcal{V}$.

There are two natural choices present in Shannon's approximations: letters or words. Words are intuitive because they carry meaning, but a word-level vocabulary is large and grows with every new word added to the language. Letters are compact, but a sentence would contain about 5x more tokens. Furthermore, this requires the

model to learn spelling before semantics. A practical middle ground, and the most common choice in modern systems, is *subwords*: units like "walk" and "ing" that are combined from characters based on frequency in a training corpus. This keeps the vocabulary manageable and the sequences reasonably short.

Tokenization Example:

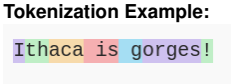

Figure 2: A visualization of sub-word tokenization. `https://platform.openai.com/tokenizer`

When Shannon constructed his orders of approximation, there were no large text databases or even computers to count over millions of documents. Instead, Shannon employed a manual simulation of probability by skimming through novels to find a specific word, recording its immediate successor, and then hunting for the next occurrence of that new word to repeat the chain. Shannon did not continue past second order approximations. He did write in his report that "it would be interesting if further approximations could be constructed." This is the same core idea underlying today's large language models. The task is **next-token prediction**: given all tokens so far, assign a probability distribution over what comes next, and then sample from it. In what follows, we will discuss important technical details like word embeddings, attention, and Transformers. In some sense, these are all just in service of doing next-token prediction as well as possible.

Part 2: Word Representations

Given a vocabulary $\mathcal{V}$, how should we represent each token? They are discrete elements from a finite set, so a natural idea is **one-hot encoding**: a $|\mathcal{V}|$-dimensional vector with a 1 at the token's index and 0s elsewhere. But there are some problems with this approach. First, there is *no notion of similarity*: every pair of tokens is equidistant. Second, there is extreme *sparsity*: a vocabulary of 100,000 tokens yields 100,000-dimensional vectors that are almost entirely zeros. Third, there is *no semantic structure*: "king" and "queen" are no more related than "king" and "banana".

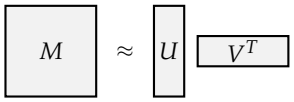

Figure 3: Matrix Factorization.

It would be much better to have dense, semantic vector representations. To get there, we will take a detour through some related ideas that developed in applications of information retrieval and personalized recommendation. In the late 80s, Latent Semantic Analysis (LSA) applied singular value decomposition (SVD) to the term-document co-occurrence matrix to find a low-dimensional latent space where semantically similar words and documents end up near each other.[1] SVD gives you embeddings, but it is expensive to compute and difficult to update as data grows.

The same idea resurfaced in recommender systems. In 2006, Netflix released a dataset of 100 million movie ratings and offered $1 million to whoever could best predict how a user would rate a movie

[1] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6), 391–407.

they hadn't seen. A key insight that emerged was to learn embeddings via *stochastic gradient descent*. Unlike the SVD used in LSA, which requires a complete matrix, SGD allowed researchers to perform *matrix factorization* scalably on sparse data. We assign each user a vector $u_i \in \mathbb{R}^k$ and each movie a vector $v_j \in \mathbb{R}^k$, and score their compatibility as a dot product $u_i^\top v_j$.

While the competition was ultimately won by a massive ensemble of models,[2] matrix factorization became the dominant paradigm for collaborative filtering recommendation due to its scalability and effectiveness at capturing latent preferences. This approach remained effective even in the absence of explicit ratings. By leveraging *implicit signals*, such as viewing history or clickstreams, researchers could model the probability distribution of user behavior. The objective was to learn embeddings where a high dot-product similarity between a user vector and a movie vector indicated a high likelihood of a future "interaction."[3] This shift from explicit scores to observed co-occurrence is precisely what powers modern language models. Just as we don't need a user to "rate" a movie to understand their tastes, we don't need a dictionary to define a word to understand its meaning. We only need to observe its neighbors.

[2] The winning team, BellKor's Pragmatic Chaos, blended over 100 models.

[3] Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets.

The key insight behind learning word embeddings is the distributional hypothesis: *similar words appear in similar contexts.* To borrow a classic example from science fiction author Ursula K. Le Guin, consider the word *ansible*. If you encounter sentences like "she sent a message across the star system via *ansible*" and "the *ansible* allows for instantaneous communication between distant colonies," you don't need a technical manual to understand the concept. Even if the physics are fictional, the statistical neighborhood forces the vector for *ansible* to settle near "radio," "telephone," or "telegraph."

Word2Vec is an operationalization of this idea published by researchers at Google in 2013.[4] In the *skipgram* formulation, the goal is to learn embeddings such that a target token and its context tokens score higher than random tokens. For a target token t_i, define its **context** as the window of w tokens to its left and right:

[4] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, 26.

$$s = (\ldots, \underbrace{t_{i-w}, \ldots, t_{i-1}}_{\text{left context}}, \; t_i, \; \underbrace{t_{i+1}, \ldots, t_{i+w}}_{\text{right context}}, \ldots)$$

We define a center embedding $v_i \in \mathbb{R}^d$ and a context matrix $\mathbf{U} \in \mathbb{R}^{|\mathcal{V}| \times d}$ where each row u_j is a context vector. The probability distribution over the vocabulary is:

$$P(\cdot \mid t_i) = \text{softmax}(\mathbf{U} v_i), \quad \text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{j=1}^{|\mathcal{V}|} \exp(z_j)}$$

We minimize the *negative log-likelihood (NLL)* over T tokens:

$$J(\mathbf{U}, \mathbf{V}) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-w \le j \le w, j \neq 0} \log P(t_{t+j} \mid t_t)$$

The exponentiated negative log-likelihood $e^{J(\mathbf{U},\mathbf{V})}$ is the **perplexity**. It represents the model's uncertainty: a perplexity of 1 indicates perfect prediction, while a perplexity of $|\mathcal{V}|$ is equivalent to uniform random guessing.

The result is that every token gets a learned d-dimensional embedding vector v_i. Despite receiving no explicit supervision about meaning, the resulting embeddings organize concepts geometrically. We can observe that gendered pairs (king/queen, duke/duchess) form parallel directions in the space, with e.g., vec("king") − vec("man") ≈ vec("queen") − vec("woman"). Morphological variants (tall, taller, tallest) form consistent geometric progressions. It is interesting that these linear relationships emerge purely from co-occurrence statistics.

One downside of this approach is that, once trained, word2vec embeddings are static. The token "bank" has the same vector whether it appears in "river bank" or "bank teller." To truly capture the distribution of language, we need dynamic embeddings: representations that shift with context.

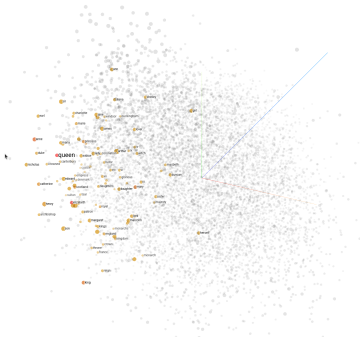

Figure 4: Visualization from `https://projector.tensorflow.org/`

Part 3: The Transformer

By the late 2010s, the deep learning revolution was well underway, and word embeddings were well established. As a result, we were ready to take the prediction game seriously as the primary objective, and build a model powerful enough to do it well over long sequences. The supervised learning task is next-token prediction: given all tokens so far, predict the next one. In particular, we need to learn a conditional distribution of the next token given the past tokens. In contrast to skipgram, we need to condition on the entire prior sequence, rather than just one token. How can we model the fact that meanings of tokens shift depending on those that have come before?

The key idea for modeling context dependence is a mechanism called **self-attention**. We draw inspiration from a search engine: given an input (query), rank all documents (keys) by similarity, and return a weighted combination of their contents (values). Self-attention applies this idea within a sequence. For a query token (e.g., "bank"), we compute its dot-product similarity with every other token in the sequence (the keys), then use those similarities as weights to

form a new embedding:

$$\text{new\_vec("bank")} = \sum_j \underbrace{\text{softmax}(\langle \text{qvec("bank")}, \text{kvec}(t_j) \rangle)}_{\text{attention weight}} \cdot \text{vvec}(t_j)$$

The result is an embedding for "bank" that depends on everything else in the sentence. "Teller" pushes it toward the financial sense; "river" pushes it toward the geographical sense. We may use slightly different vector representations of a token depending on whether we are treating it as a query (qvec), a key (kvec), or a value (vvec). In what follows, we make this idea precise by defining the **attention block**, the key component of the Transformer architecture.

First, we discuss how to define the vector embeddings. Since attention treats all positions symmetrically, we also add **positional encodings** to break symmetry. Each token's d dimensional input representation is:

$$\text{vec}(t_i) = \text{tok}(t_i) + \text{pos}(i)$$

where $\text{tok}(t_i)$ is the learnable token embedding and $\text{pos}(i)$ encodes position (learned, absolute, or relative). Then we define linear transformations of the raw embeddings so

$$\text{qvec}(t_i) = W_Q \text{vec}(t_i), \quad W_Q \in \mathbb{R}^{d_k \times d}$$

and similarly for kvec and vvec. The dimension of qvec and kvec is d_k and the dimension of vvec is d_v.

The model input is a sequence of n tokens. In matrix form, we define the input $X \in \mathbb{R}^{n \times d}$ as the sequence of these position-aware embeddings, where row i is $\text{vec}(t_i)$. The query, key, and value matrices can therefore be defined as different linear projections of X

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V\,.$$

Since the task is next-token prediction, a token should only be able to attend to tokens that came before it and not to future tokens it is supposed to predict. We enforce this by applying a **causal mask** to the attention scores, zeroing out any position where a query attends to a future key. This ensures the model cannot "cheat" by looking ahead. We incorporate a causal mask M into the computation:

$$O = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right) V$$

where M is a mask matrix with 0 on or below the diagonal and $-\infty$ above it. In code:

```
def causal_softmax_attention(Q, K, V, mask):
    # mask is a boolean matrix where True means "can attend"
    scores = (Q @ K.transpose(-2, -1)) / math.sqrt(Q.size(-1))
    scores = scores.masked_fill(mask == 0, float('-inf'))
    return torch.softmax(scores, dim=-1) @ V
```

A full **Transformer block** wraps attention with layer normalization, a position-wise feed-forward MLP, and residual (skip) connections ($x + f(x)$) for stable gradients. The Transformer architectures powering today's LLMs stack many of these transformer blocks together. For example, the minimal version of nanoGPT consists of 3 attention blocks, while GPT-3 scales this to 96 attention blocks and 175B learnable parameters.

Note on Scale: Modern architectures scale d to keep model capacity high, but the $n \times n$ attention matrix is the primary bottleneck. As context windows reach millions of tokens, the $O(n^2)$ memory requirement exceeds standard GPU VRAM, necessitating optimizations like FlashAttention which avoids materializing the full matrix.

Next-token prediction is a deceptively simple objective. Many classification problems can be framed as completing a sentence:

> Given this review: `<REVIEW>`, is the sentiment positive? Answer:

The model completes with "yes" or "no." The same trick works for translation, summarization, question answering. Almost any task in language can be cast as autocomplete. This is why next-token prediction, despite being the same game Shannon played with Mary in their living room, turns out to be such a powerful and general training objective.

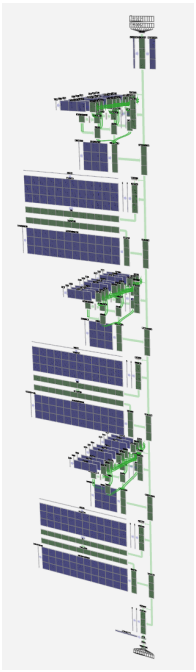

Figure 5: NanoGPT visualization from `https://bbycroft.net/llm`

References

The material in this lecture was developed with guidance from

- Gangavarapu, Tushaar. "Lecture 24: Language modeling and Transformers." *Cornell University CS 3780/5780 Notes*, Spring 2025. `https://www.cs.cornell.edu/courses/cs3780/2025sp/notes/lec24-llm.pdf`
- Gibson, Richard Hughes. "Language Machinery: Who will attend to the machines' writing?" *The Hedgehog Review*, Fall 2023.