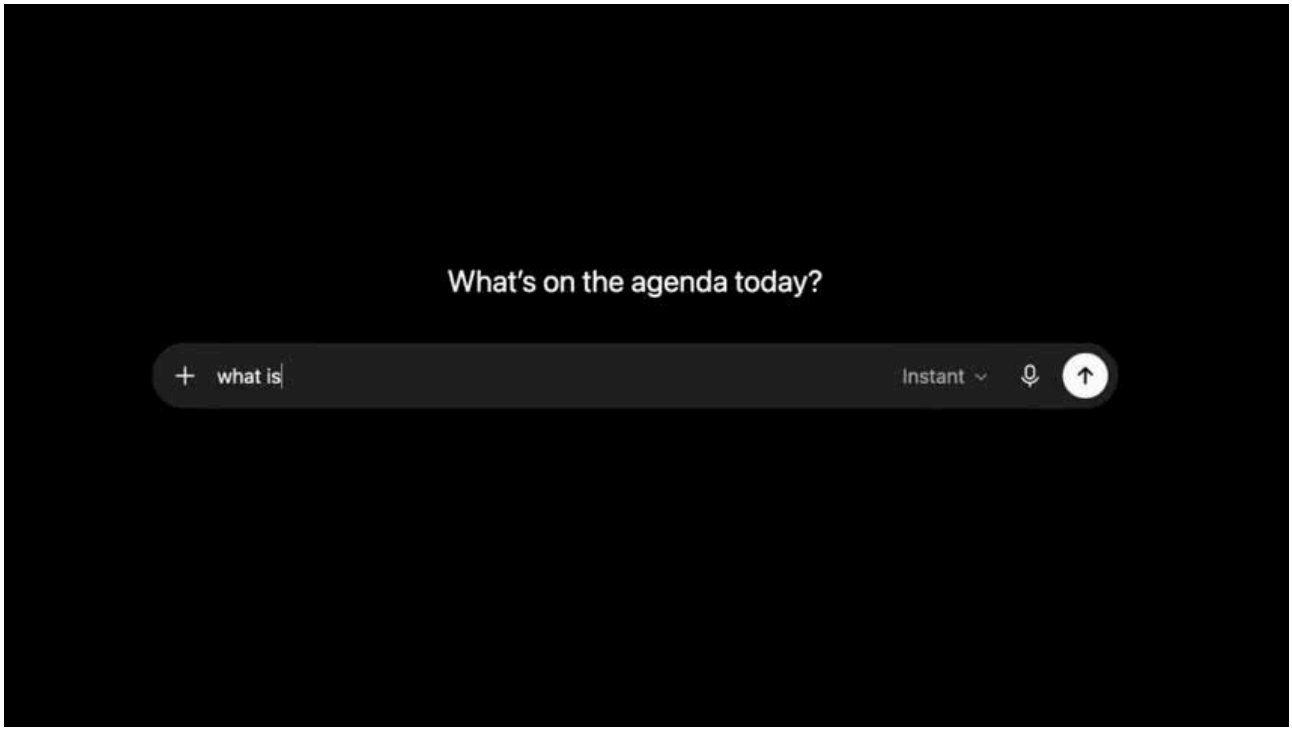
# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

## Introduction

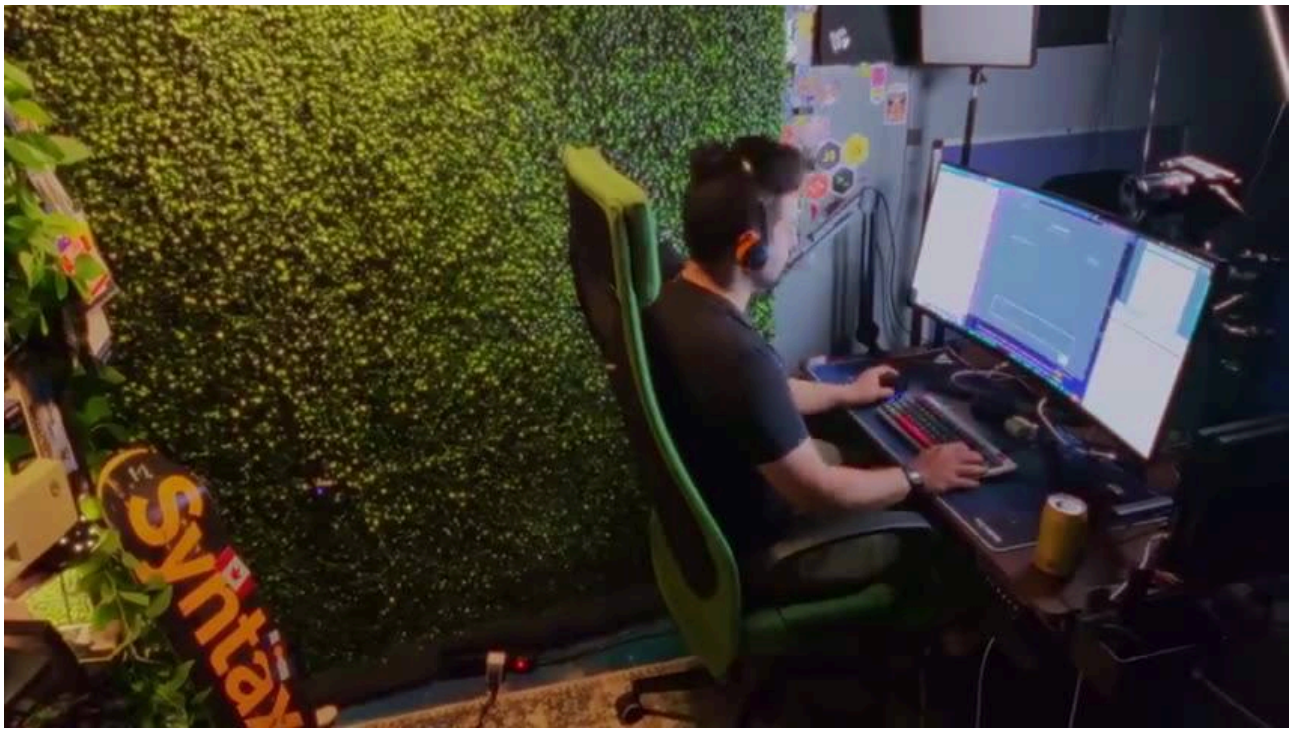



at 0:03

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0



at 0:15

What happens when you type a prompt into an LLM like ChatGPT, Claude, or Gemini and press enter? The answer, as it turns out, involves over 80 years of research, billions of dollars, and some of the most elegant math humans have ever invented.

To answer this question, I built an LLM from scratch, and I'm going to walk you through that full journey, from keystroke to streamed response, with working code at every step.

## Claude Shannon and the Origins of Information Theory

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 0:53

Our story actually starts back in 1950. Claude Shannon, the father of information theory, sat down with his wife Betty to play a game. He showed her a passage of text with the next letter hidden, and she had to guess what came next, letter by letter. He measured how often she was right. But what he was really measuring was how predictable written language is.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 1:01

at 1:10

His finding was that, given enough context, the next letter is often nearly certain. This means language

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

has deep statistical structure, and over 70 years later, that's pretty much what an LLM is doing: predicting what is coming next.


at 1:23

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 1:28

LLMs are statistical models of language. They extract patterns and relationships from massive amounts of text and build a mathematical representation of how words relate to each other. People often describe them as next-token predictors, and while that's true, there is a massive amount of machinery involved before you can get to the point of predicting what comes next. That's really what I wanted to understand.

We are increasingly interacting with these AI services on a daily basis, whether at home or at work, and a lot of the details are hidden from us. Most people just hand-wave over it: "Oh, it predicts the next token." I wanted to dive deeper and have a better understanding of all this stuff. So that's exactly what I'm going to break down in this video. You're going to walk away with a deeper understanding of how LLMs work.

If that sounds good, let's dive in. My name is CJ. Welcome to Syntax.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 2:02

## Early Chatbots: Turing, Eliza, and Perry

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 2:14

Before LLMs existed, we had chatbots of various forms over the past 60-plus years. This really all started back in 1950, when Alan Turing published his paper "Computing Machinery and Intelligence." This is where he proposed the famous Turing test, which asked: if a machine can converse indistinguishably from a human, can it think? This really set the target for the next 70 years.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

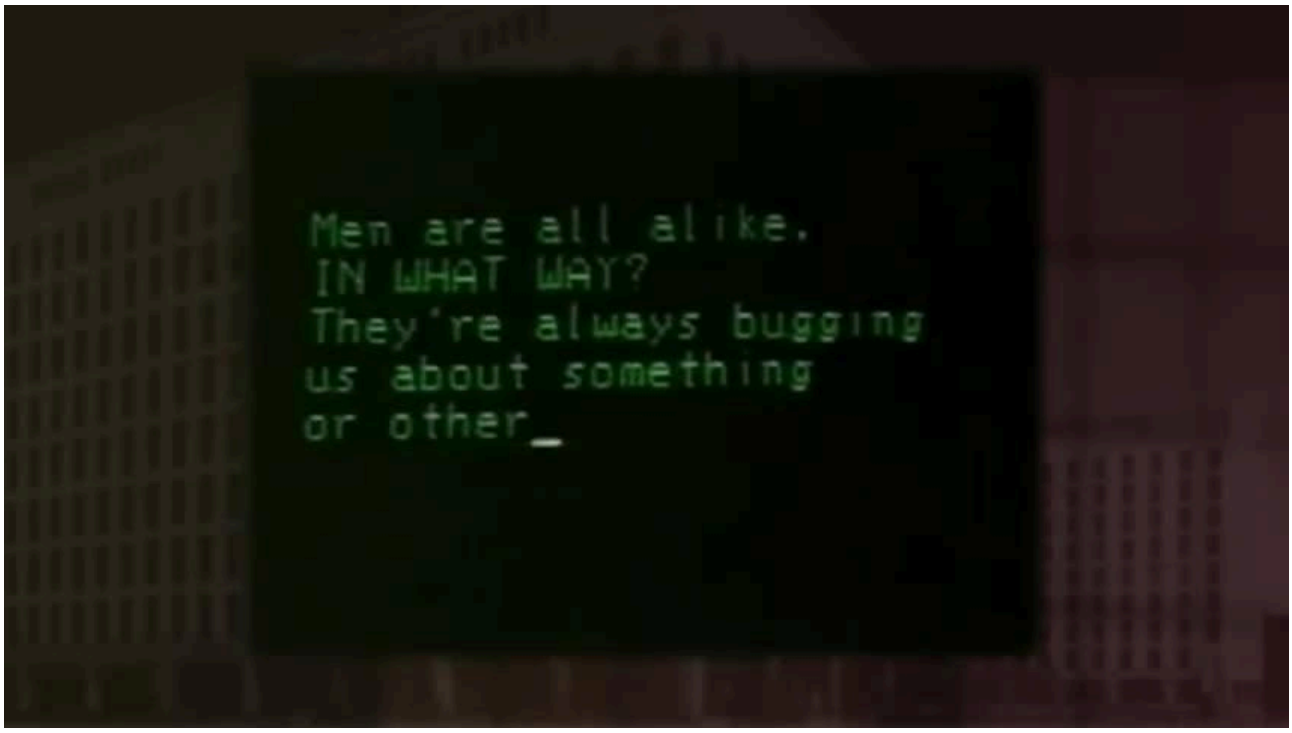

at 2:35

In 1966, Joseph Weizenbaum at MIT built a program called Eliza, which was essentially a pattern-matching program that mimicked a Rogerian therapist. It had no understanding whatsoever. Essentially, whatever the user typed, it would go through a series of if-statements to determine how it should respond.

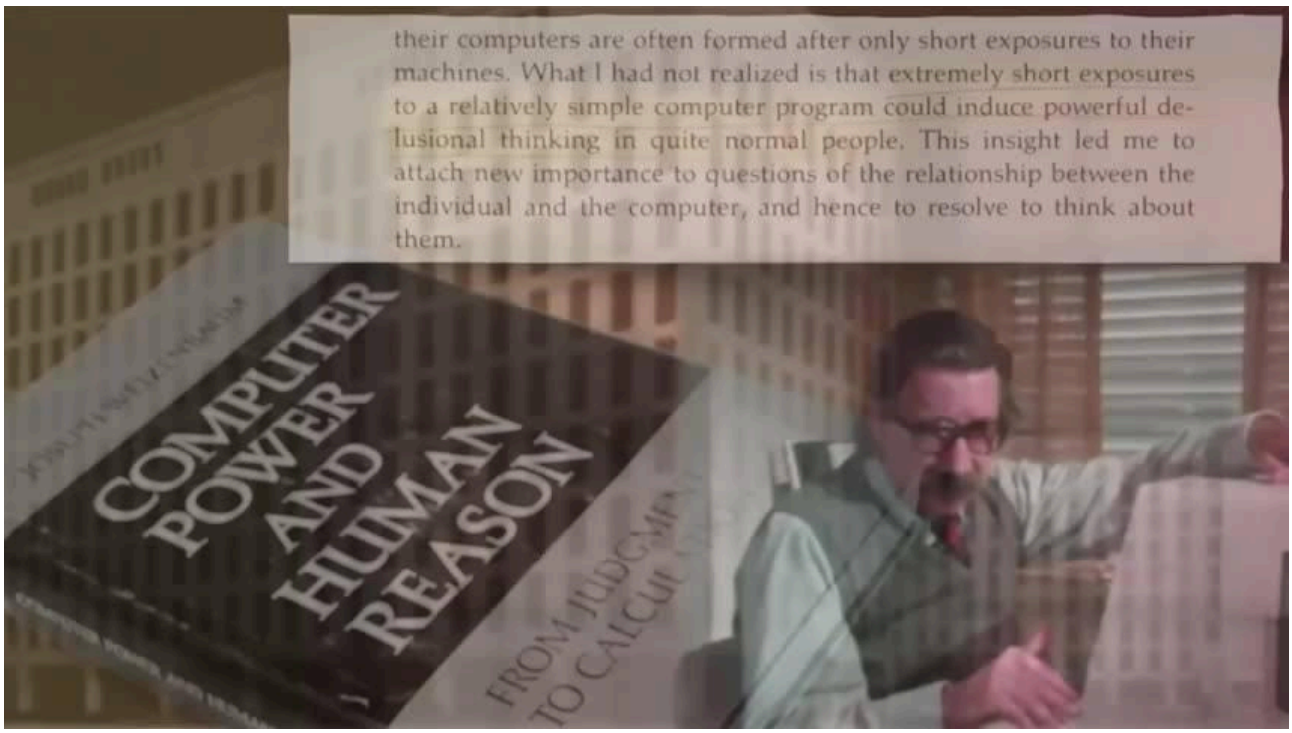
# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0




at 2:49




at 2:54

A program like this is known as a rule-based system. A programmer has to manually write out all the

statements to detect, based on what the user typed, how the program should respond. It has no built-in understanding.

at 3:03

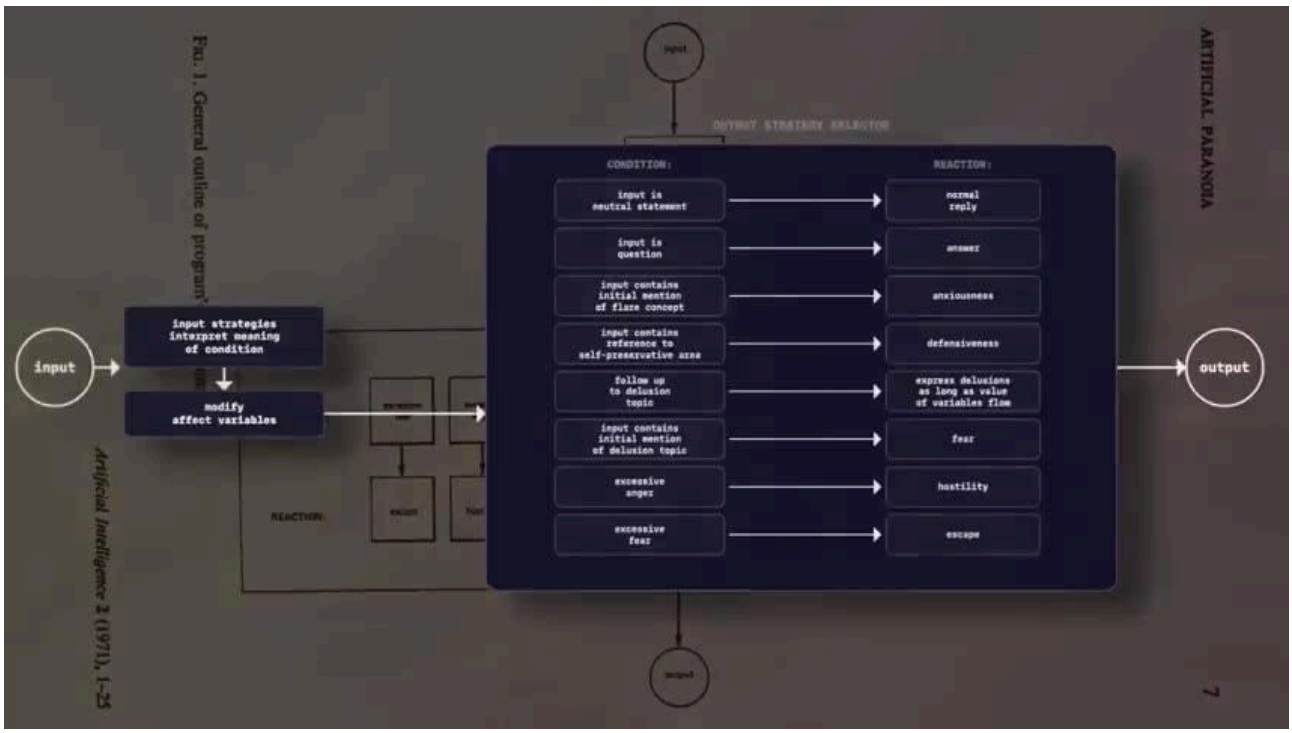
Famously, Weizenbaum's secretary was using the program and actually asked him to leave the room so she could talk to Eliza privately. He later wrote: "I had not realized that extremely short exposures to a relatively simple computer program could induce powerful delusional thinking in quite normal people." We're seeing this today too — people making comments about how dependent they are on talking to ChatGPT, as if it's their friend or their therapist, even though there's not a real human on the other side.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 3:12

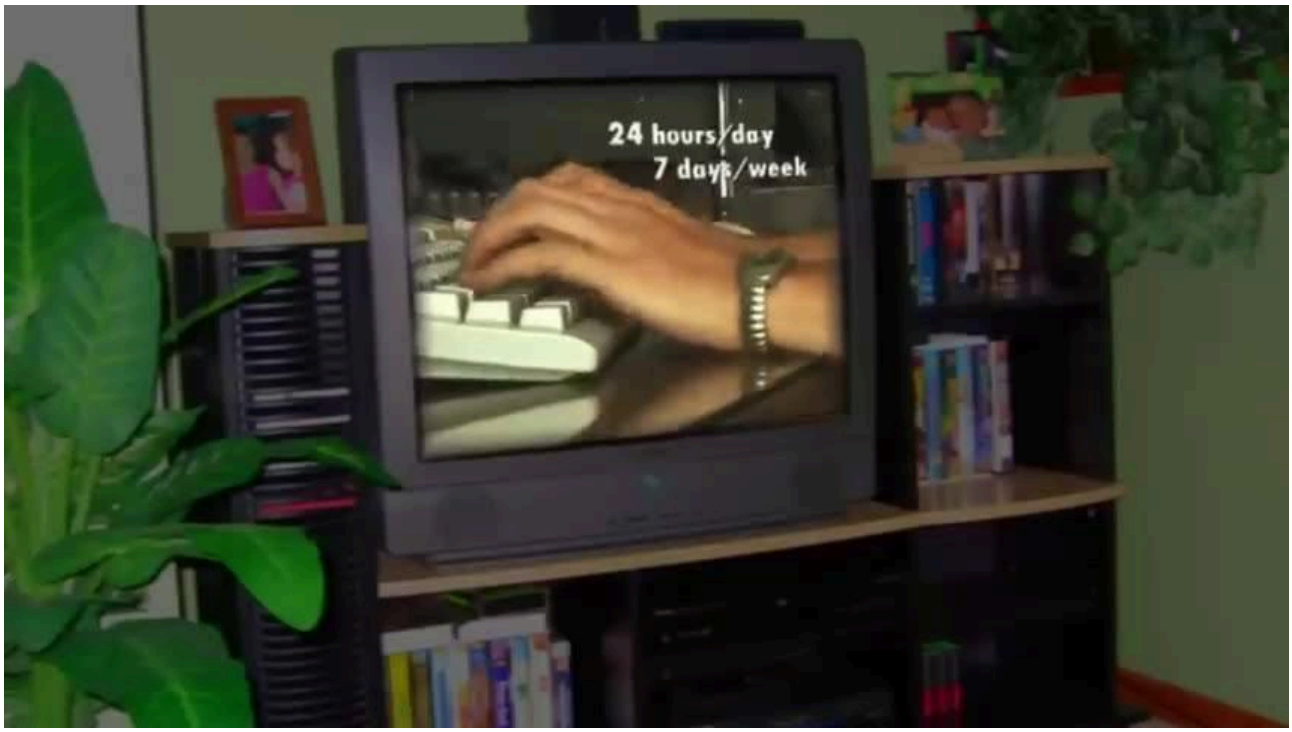



at 3:20

Later, in 1972, Kenneth Colby at Stanford built a computer program called Perry, which simulated a

paranoid schizophrenia patient. It had internal states for anger, fear, and mistrust, and it shifted its responses accordingly. Psychiatrists who sat down to talk to this chatbot couldn't distinguish it from real patients. Later that year, Eliza and Perry were actually connected together — you essentially had a therapist chatbot talking to a patient chatbot.

## Decades of Rule-Based Systems




at 3:46

After that, there were decades of rule-based systems. Alice was released in 1995, with 41,000 handwritten patterns. SmarterChild, which some of you might remember, was released in 2001 on AOL Instant Messenger. This was one of the first chatbots millions of people used daily, and it could be seen as a precursor to ChatGPT — kind of a do-everything chatbot.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 4:00

Every single one of these chatbots, up until this point, was based on rules, scripts, templates, or decision trees. A human had to sit down at a computer and program every single response in advance. So the jump from Eliza to ChatGPT isn't just a better chatbot — it's a completely different mechanism.

## Building a Simple Rule-Based Chatbot

Before we dive into how LLMs work, let's take a look at how a simple program like Eliza could be written.

This is the simple chatbot I have set up here. If I say "hello," it will respond just like an LLM: "Hello, how can I help you today?" Or if I say "hi," it will respond the same way. All of the code for everything I'm going to show you in this video is linked in the description, and we're going to start here with this simple chat.

You can see I have a list of every possible greeting, and if the user types any one of these greetings, it responds with that exact response: "Hello, how can I help you today?" We have one function that handles all responses. The user's message is passed in, we replace all the special characters, and remove any white space. From there, we basically have a list of words corresponding to the user's message, and the first thing we check is whether one of the greetings we've defined is in the user's message. If it is, we immediately respond with "Hello, how can I help you today?"

If you were to just come across this chat UI and say "hi," you might initially think there was an LLM on the other side, because it responds in a very similar way to how ChatGPT responds.

The next thing we do is look at the user's message to determine how we can respond in various other

ways. The first check is: does the user's message start with "I feel"? I'll say, "I feel happy today." It responds, "Why do you feel happy today?" We extract the thing after "I feel," replace any "I"s with "you"s — so if the therapist is responding, they say "you" instead of "I" — and then it simply responds with "Why do you feel [the extracted feeling]?" In this case it extracted "happy today," so it said, "Why do you feel happy today?"

I can say something like, "My boss gave me good remarks." Then it says, "Tell me more about your boss." In the code, we have a matcher for "my." If the user's message contains "my," they're talking about some subject. We extract that subject and respond with, "Tell me more about your [subject]." In this case, the subject is "boss."

As you can see, the code is super straightforward, but if you didn't know what was going on behind the scenes, you might think there was something smart on the other side.

Beyond these matchers, we also have catch-all responses — random continuations like "Please go on," "Tell me more about that," "How does that make you feel?" If I say something we didn't match against, like "He is cool and is a breakdancer," it will respond with "Please go on." We didn't have any direct matchers for that particular sentence, so it picked a random continuation, and the user can keep chatting with the bot. With just a few if-statements, we can make the user feel like there's something smart on the other side.

## The Black Box Model

So we built a super simple chatbot — it's just a bunch of if-statements — but the UI the user is interacting with is almost exactly the same as ChatGPT or Claude. It has an input box; the user types, and gets back some response.

I like to abstract this and think of it as a black box: it has some input, performs some process, and gives some output. In this case, the process is just running the input through a bunch of if-statements. What's interesting is that we can actually replace that box with an LLM: a prompt goes in, the LLM does something, and an answer comes out. That's essentially what we're going to do throughout this video — slowly replace the pieces to get smarter and smarter machinery.

I like to think about the world of computing and technology this way. Everything is initially a black box, but we can start to uncover how that thing works. And if we generalize it as inputs and outputs, we can even replace the black box with smarter machinery later on.

Now, before you can prompt an LLM, that LLM needs to be created.

## What Happens When You Ask ChatGPT a Question

When you type a question into ChatGPT, that gets sent to a pre-trained model. That training process has to happen ahead of time, and essentially it produces a model file that then lives on a server somewhere — that's actually what we're interacting with. But I'm already getting ahead of myself, because first we need to talk about what a model is.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

LLM stands for large language model — it's actually crazy that I haven't defined that this far into the video. It has the word "model" in there, and a model is a neural network: a system of interconnected nodes called neurons that are organized into layers. Data flows through the input layer, passes through one or more hidden layers, and comes out through the output layer. Each connection has a weight, essentially a number that controls how much influence one neuron has on the next.


at 8:34

## A Brief History of the Neural Network

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 8:50

The concept is fairly simple, but the history is wild, and it starts back in 1943. Warren McCulloch was a neurophysiologist, and Walter Pitts was a self-taught logician. Together they published a paper proposing that the brain's neurons could be modeled as simple on-off logic gates — connect enough of them in the right configuration, and they could, in theory, compute anything. That was just a mathematical thought experiment; nobody could actually build it yet.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 9:18

Fifteen years later, in 1958, a psychologist named Frank Rosenblatt actually built one, and he called it the perceptron. It was a physical machine — a room-sized contraption of wires, motors, and photocells that could be shown images on cards and learn to classify them. The Navy held a press conference, and the New York Times reported it as "the embryo of a computer that would be conscious of its existence."

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 9:29

Then, in 1969, two of the most prominent figures in AI, Marvin Minsky and Seymour Papert, published a book called *Perceptrons*. They proved that a single-layer perceptron couldn't solve certain basic problems — the most famous being exclusive or (XOR): given two binary inputs, output a one if exactly one of those inputs is a one. That's trivially simple for a human and mathematically impossible for a single-layer perceptron.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 9:50

Multi-layer networks *could* solve XOR, but nobody knew how to train them effectively. That breakthrough came in 1986, when David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper in *Nature* describing backpropagation — a method for training multi-layer networks. Essentially, you make a prediction, measure how wrong it is, and then propagate that error signal backward through every layer, adjusting each weight to make the prediction slightly less wrong. You repeat this billions of times.

Every neural network you interact with today — whether it's Claude, ChatGPT, or Gemini — is trained with some variant of this backpropagation algorithm.

## Training a Neural Network to Solve XOR

Let me show you the code. I'm going to use the exact same problem that killed the perceptron: exclusive or. This example is called XOR Neural Net, and it trains a neural network in real time to solve XOR.

If I pass in "single layer," this trains a single-layer network, and you can see that even after 5,000 iterations, it doesn't reach the correct output we're expecting. But if I pass in "multi-layer," it does reach a point where, given these inputs, we get our expected output. Even around 800 iterations, the loss is extremely low, and it just keeps getting lower from there — by 5,000 iterations, the loss barely improves any further.

After all those iterations, we end up with weight values for our neural network such that, given two inputs — false/false, false/true, true/false, or true/true — we get the expected outputs. It isn't perfect,

but we round these values: this rounds down to zero, this rounds up to one, up to one, down to zero.

The main thing I want you to see in the code is that after we train these neural networks, it actually creates a file in the .data folder that contains the weights for that neural network. So when we talk about a model, it's literally just a file with weight values inside of it.

## Reading the Weight Files

For the single-layer network, we're connecting those two inputs to one single output. So we need a connection from the first input to the output and from the second input to the output — each connection has a weight, which is why we see two weight values. The final calculation on the output node also includes a bias value. So the single-layer neural network is literally just three numbers.

If we look at the multi-layer weights, it's a little more complex. This neural network has four neurons in the hidden layer, and we need to connect each of the inputs to each of those four neurons — that's why we see two arrays of length four. The first array holds the weight values for the first input, since it needs to be connected to each of those four neurons (four connections), and the second input needs the same — four more weight values. Then there are four bias values, one for each neuron.

Those four neurons in the hidden layer then connect to the single output, which is why we see one last set of four weights — four connections from the neurons to the output, plus a bias value. So for this very simple neural network, the model file is just a bunch of weight values. This example is trivial, but every neural network works exactly this way — every one you come across will have a weight file, a model file containing all the weights calculated after backpropagation training.

## Walking Through the Code

If we look at the code, we have our inputs and our target outputs. Every neural network has a certain number of inputs and a certain number of outputs. Ours has two inputs — false/false, false/true, true/false, or true/true — and one output. The expected outputs are: false and false is false, false and true is true, true and false is true, and true and true is false. Every neural network needs this: what are your inputs, and what are your expected outputs given those inputs.

We have a train_single_layer function that takes in the number of epochs. In the output, we used 5,000 total epochs, which is what we default to — that's the number of iterations where we calculate our current outputs, figure out how far off we are from the expected outputs, adjust the weights, and repeat. In this example we always do 5,000 iterations; you could also set it up to keep going until the total loss is within a certain threshold, but here it's a fixed count.

You can see that the two weight values we're trying to calculate start off as completely random values. For each iteration, we calculate the overall value using the weights, the inputs, and the bias, giving us an output for that calculation. We then compare it to our target, calculate the error, and use calculus — the derivative — to figure out the delta, how far off we are from the expected output. Then we adjust all the weights and the bias accordingly. After 5,000 iterations, we end up with calculated weight and bias values and can determine the overall loss.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

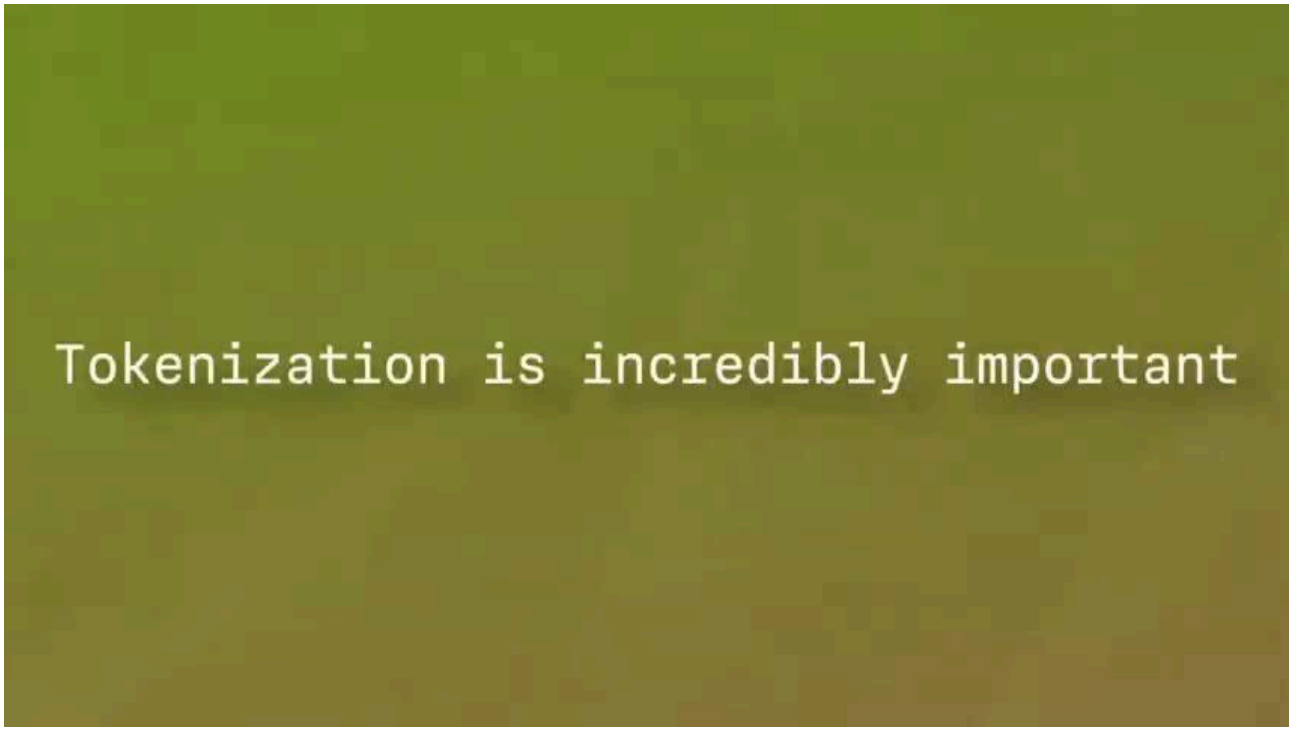
train_multi_layer is very similar, except we have to come up with random weight values for every connection. The iteration code is very similar too, except we work through two layers: first from the input to the hidden layer, then from the hidden layer to the output. So the code is very similar overall.

## Multi-Layer Neural Networks

Now, we just have two different deltas: the output delta, which is how far we are off from the output to our expected values, and the hidden delta, how far we are off from that hidden layer. From there we update all of the weights accordingly and then repeat. The code from single layer to multi-layer isn't that much more complex — it's really just calculating between those layers, and we can have any number of hidden layers we want. The code will be very similar.

That's the basics of a neural network. You set up your inputs, you set up your outputs, set up your hidden layers, and then just start iterating to adjust those weights until you reach a point where those weights give you the expected output given those inputs.

## From Prompts to Inputs




at 16:03

Okay, so we've got the basics of a neural network. As we showed, neural networks have inputs and outputs. Essentially, the prompt that you type into your LLM is going to be an input into that neural network. But the neural network doesn't just accept a block of text, doesn't just accept your question — that block of text needs to be broken down so it can be turned into inputs for that neural network. The first step in that process is known as tokenization.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

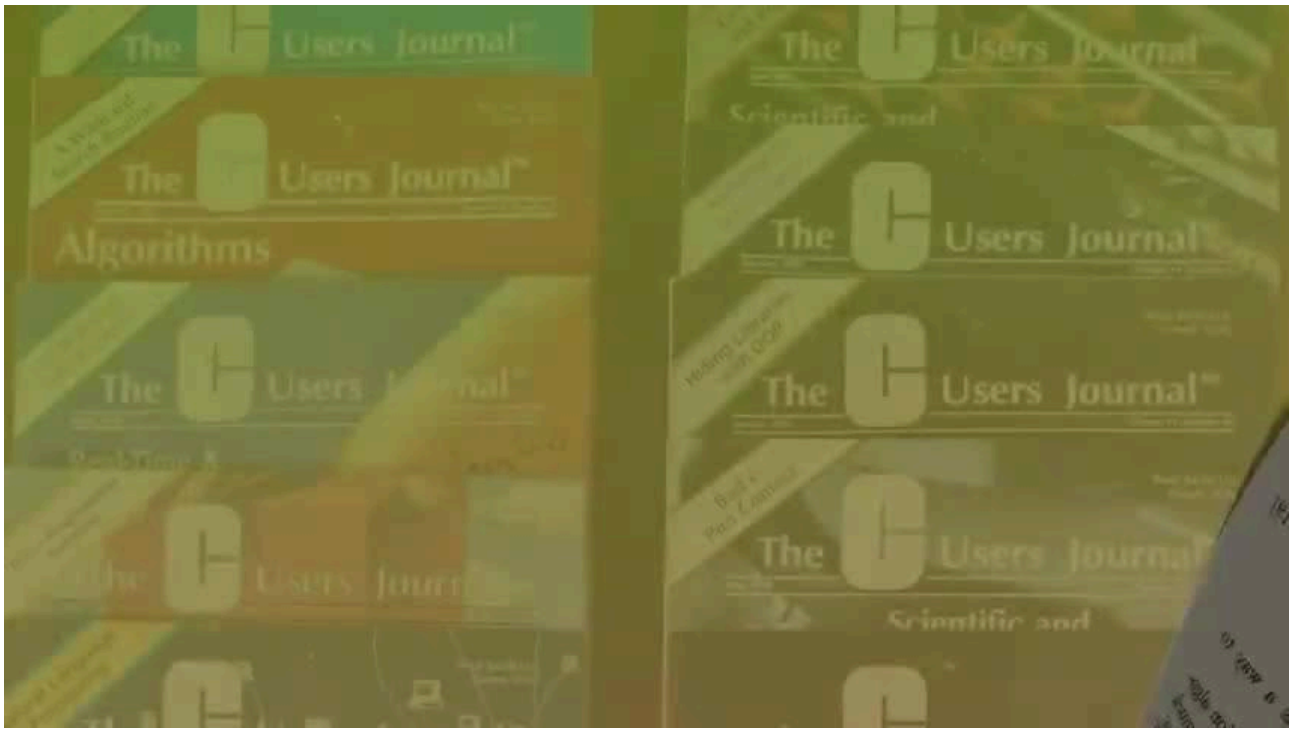
at 16:32

## Tokenization

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0




at 16:37

A model breaks your prompt into pieces — not words, not characters, something in between. A token is the smallest unit of text a language model works with. The word "the" is one token. The word "tokenization" might be split into "token" and "ization," two tokens. A space is often part of a token. A new line is a token. An emoji might be multiple tokens. The model doesn't see words the way you do — it sees tokens.

How does it decide where to split? It uses an algorithm called BPE, or byte pair encoding. BPE was invented in 1994 by Philip Gage as a data compression technique — it had nothing to do with language models. The idea was simple: look at a sequence of bytes, find the pair that appears most frequently, replace it with a new symbol, and repeat. Essentially, it compresses data by finding common patterns.

Twenty-one years later, in 2015, researchers Rico Sennrich, Barry Haddow, and Alexandra Birch at the University of Edinburgh realized this same algorithm was perfect for building vocabularies for neural machine translation. Instead of deciding in advance what all the words are, or what your vocabulary is, you let BPE learn a vocabulary by iteratively merging the most common character pairs in your training data. So common words like "the" stay whole, and rare words get split into sub-word pieces. The beauty of this is that one algorithm handles English, Japanese, Python code, TypeScript code, emojis — all without language-specific rules.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 17:34

Tokenization isn't just a pre-processing detail — it has real consequences. Token count determines cost: every API call is priced per token. Token count also determines what fits into the context, and every model has a maximum context window, measured in tokens. If you exceed it, something's going to get cut.

## Tokenization Code Walkthrough

Let's take a look at the code for BPE tokenization. This demo is called the basic tokenizer. You can drop in some text and it will show you how it broke that training text down into tokens. With "the cat sat on the mat," you can see at each merge the token pairs it came across, and then finally we get our overall vocabulary. This came across six unique tokens total.

The main idea with BPE is pair merging — the most frequent pairs are merged. With a very simple training text like one sentence, it's going to find each word as a unique pair. But in the code, we can specify a maximum number of merges to do. If we change this to something like three and train it on the same small bit of text, you'll see it actually finds "at" as a unique token, because "at" appears multiple times in the training data — "cat," "sat," "mat." And because the word "the" appears twice, it got its own token, and everything else was just individual letter tokens.

If we dive into the code, one of the first things to look at is the regular expression. Every tokenization algorithm first runs all the training data through this regular expression to split it up, because BPE never merges across word boundaries. Even before we start doing the merges, we need to define ahead of time what the whole groups are that we're going to use to find the individual merged elements within them. In

this case, our groups are words.

If you look at the algorithm for GPT-2 or GPT-4, they have a predefined, really complex regular expression, because they've defined rules up front for how to split things ahead of time. But our initial step here is just to split on whole words — the things we're going to merge are the individual words, split on spaces.

The next step is actually an optimization: we determine the frequency of all those words. Our regular expression splits on spaces, and then for every unique piece of text, we count the number of occurrences. That count gives us a weight for how much we care about that token in the training data. So in "the cat sat on the mat," "the" appears twice, giving it a higher weight than the other words.

Then we get into the actual training algorithm. We iterate up to our max number of merges — I showed earlier this set to 10,000. You get to decide ahead of time how much iteration you want, and that determines how large your vocabulary gets. With a really large max merge size, we're more likely to find all the unique tokens in a given training set.

Then we have the bulk of the work: this looks at every character pair to find the most commonly occurring ones, taking into account the weight — how often that particular word occurs. We find the most common occurring pair with the highest weight, and that becomes a new piece we merge on in the next iteration. We take that best pair, merge all the groups — in this case merge all the words together — and then repeat to find the next most commonly occurring pair.

To see a more interesting example, I'm going to plug the *Bee Movie* script into this tokenizer, and we can watch it do all the merges and find all the unique tokens in the *Bee Movie* script. After training, this found 2,088 unique tokens, essentially all the unique words in the script — you can see it found all the whole words.

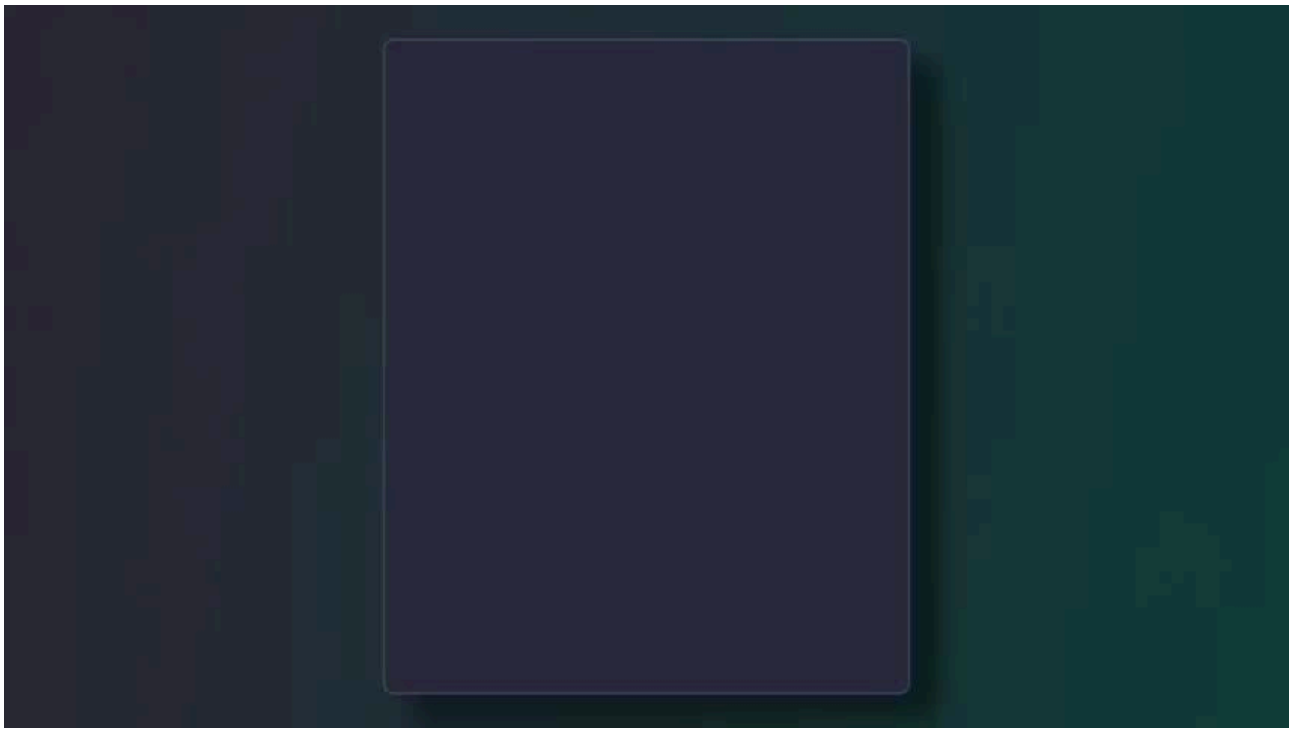
But if we reduce our max merges to something like 1,000 and try this again, we'll see tokens in our vocabulary that are essentially broken-up words. You can see the word "according" got broken into four tokens, because we only have a certain token budget, and the word "according" didn't appear that many times in the overall *Bee Movie* script.

That's the basics of tokenizing.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0



at 21:40

## Embeddings

We've analyzed a large dataset, extracted all the possible tokens using the BPE algorithm, and that gives us a vocabulary where each token has a numeric ID — essentially the index of that token in the vocabulary. These IDs are just arbitrary numbers; they don't tell us anything about the actual meaning of those tokens or how they relate to each other. So we need to turn those tokens into something the model can actually reason about.

The idea behind that goes all the way back to 1884, when logician and philosopher Gottlob Frege coined the context principle, which states: "Never ask for the meaning of a word in isolation, but only in the context of a proposition." Seventy years later, in 1957, British linguist J. R. Firth put it more memorably. He wrote: "You shall know a word by the company it keeps." Today we call this idea distributional semantics, and we've built it into the machines. Every modern language model begins by mapping words into a vast numerical space where neighbors share meaning — and those maps are called embeddings.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 22:27

## The Distributional Hypothesis, as Math

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 22:42


at 22:49

That one sentence, "You shall know a word by the company it keeps," is the thesis behind every

embedding model ever built. In 2013, Tomas Mikolov and his team at Google built on these ideas in their paper, Word2vec. They trained a neural network, an embedding model, to predict words from their neighbors in large amounts of text. It was a simple setup, but when they looked at the vectors the model produced, they discovered structure that no one had taught it.

For instance, take the vector for the word "king," subtract the vector for the word "man," add the vector for the word "woman," and the closest result is "queen." Nobody told the model about gender or royalty — these concepts just emerged purely from the statistics of which words appear near other words. This was essentially that first idea, "you know a word by the company it keeps," implemented as math.

## What Is a Vector?

Essentially, it's a list of numbers that identifies a point in high-dimensional space. The simplest version is a vector in 3D space, or three numbers: X, Y, and Z. Embedding vectors are much larger, though. The Word2vec paper used 300 dimensions, and GPT-3, the largest model, uses 12,288 dimensions. Each number captures some feature the model learned during training — not something a human named — but together they encode meaning as a position in space.

Words with similar meanings end up as nearby points. For instance, "happy" and "joyful" are close together, while "happy" and "refrigerator" are far apart. We can measure exactly how close two vectors are using a formula called cosine similarity, which essentially looks at the angle between two vectors. A score of one means they're identical; a score of zero means they're completely unrelated.

Modern LLM embeddings are direct descendants of Word2vec, just scaled up from individual words to entire contexts.

## Demo: Training a Word2Vec Model

Let's take a look at how to train a simple Word2vec model. This demo is called Train Embeddings, and you can pass in a couple of words, and it will train an embedding model in real time, then show you the comparison of the generated vectors between the words you passed in. For each of the words we passed in, we see the actual generated vector — the embeddings generated for each of those tokens.

The cool thing about this demo is the analogies we get from this training set. We have the classic word math of king minus man plus woman is queen, but it also works in reverse: queen minus woman plus man gives us king. There are a few other interesting examples too. In the same royalty category, prince minus boy plus girl is princess. And a fun one: kitten minus cat plus dog gives us puppy. Even with our small training set, we're able to get word math that actually makes sense to us as humans who understand language.

## The Training Corpus

The main thing you need when training a model like this is a corpus — all of the text you're training the model on. Here we have a list of a little over 100 sentences, and they're all just simple statements: "The cat sat on the mat." "The kitten is a baby cat." "She loves her pet cat." And then there's an entire section

for royalty.

The thing to note with this training data is that we're showing the model those relationships by putting both king and queen in similar contexts. "King is a man who rules." "Queen is a woman who rules." "A prince is a young man of royal blood." "A princess is a young woman of royal blood." By having those statements and just swapping out the words, the model learns that "princess" is associated with "woman" and "young," and "prince" is associated with "young" and "man." There are plenty more examples showing king, queen, princess, and prince in context so the model can learn those relationships.

## Skip-Gram: Building Word Pairs

Now let's look at the actual training. In the Word2vec paper, they talk about two different architectures: skip-gram, and CBOW (continuous bag of words). We implemented the skip-gram architecture.

The first step in training skip-gram is to create pairs of words from the training data. There's a variable called window size, which you can configure however you'd like — we set it to something sensible like five or six. Every word gets paired with every other word within that window size. For example, in the training data, take the word "king": we'd create pairs like king–is, king–of, king–man, king–who, king–rules. Then we do that with every other word too — man–a, man–is, man–king, man–who, man–rules.

We create all of these pairs, and that lets the model learn those groupings and see that, fairly often in the training data, when it sees "king," it's very often paired with "man." Or when it sees "queen," it's very often paired with "woman." Or when it sees "king" or "queen," it's very often paired with "kingdom." That's the very first step — we build up all of those pairs across all of the training data. In the run, we had 107 sentences, which created 3,970 pairs of words to train on.

## The Weights File

The other thing to look at is the actual weights file. Just like with the exclusive-or neural net, we have a file that contains all of the weights. The cool thing about Word2vec is that it's one of the simplest neural networks — essentially our embeddings are just a list of every word in our vocabulary, and the vector for that given word.

The first step was to tokenize the input training text, which we covered in the last section. You tokenize it, then create a vector — an array of numbers — for every single one of those tokens. So when you look at this embedding weights file, it's literally the entire vocabulary, and for every vocabulary word, we have a vector array.

Just like when we were training exclusive-or, we need initial random values for those vectors, so we create a randomized array sized vocabulary size times dimension. In this case, we used a dimension of 32, so every vector is length 32. You can make that dimension whatever you want — the more dimensions, the more information it can store — but here, every word in our vocabulary starts off with an array of length 32 filled with random values.

## The Training Loop

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026
https://www.youtube.com/watch?v=YmLp8qe87A0

For each epoch, we take all of our pairs of words. The first step is to push the target and the context closer together: all of the pairs of words that appear next to each other are somehow related, so our backpropagation — our way of nudging these values — says if those two words are close together, push their vectors closer together.

We also do the opposite: for every vocabulary word that never appears next to a given word in our training data, we push those vectors further apart. So on each epoch there's a push and pull — all of the words that are related get their vectors closer and closer together, and all of the words that are unrelated, that never appear next to each other, get pushed further and further apart.

## Doing Word Math with Vectors

After the model has been trained, we have vectors for every word in our vocabulary, and now we can do math with them — this is where all of the analogies you saw in the web UI come from. To do that math, we look up the vector for any one of those words. Once the model is trained, all of the vectors live in that weights file, so when we're doing inference — comparing words or doing math with them — we can literally just look up their vector values. The code says: for each of the three words we're about to do the math on, pull their vector value out of the weights file, and then do the math. Take that vector...

## From Word Vectors to Sentence Embeddings

Minus that vector, add that vector, and that gives us our resulting vector. Now, the resulting vector isn't going to be perfect — it's just a vector — but we then compare that vector to every other word in our vocabulary and choose the one that's closest. That's why you see, with queen, that was the closest vector in our vocabulary to the result of performing that math. So that's it for training embeddings.

Now, this is a very simple embedding trainer. The data set is really small, and it only creates vectors for individual words. Modern embedding models have massive data sets — literally every text written by humans in the entire history of humans. And instead of just embedding the vocabulary, they also embed statements and essentially context from all of that training data. You basically take this exact same concept and scale it up to whole sentences or paragraphs instead of just individual vocabulary words.

## What Actually Processes Tokens and Embeddings: The Transformer

So we've got tokens, we've got embeddings, but what actually processes them? The answer is a transformer. Every major LLM you might use today, whether it's GPT, Claude, or Gemini, is some form of a transformer.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 31:04

This all started with a problem. By 2016, Google's translation models were built on a type of neural network called an LSTM, short for long short-term memory. LSTMs read text one word at a time, carrying a running memory of what they've seen so far. They worked, but they were slow — each word had to wait for the previous one to finish processing.

To help with longer sentences, researchers had bolted on an add-on called attention. Attention came from a 2014 paper by Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Older translation models compressed a whole sentence into one fixed-size vector before translating. Short sentences were fine, but long ones broke. Bahdanau's idea was to let the model look back at the input at each step and focus on the words that matter right now. It worked, but it was still bolted onto the LSTM, so it was still slow.

## "Attention Is All You Need"

In 2017, Jakub Uszkoreit at Google proposed dropping the sequential part entirely and using only attention. Eight researchers built it, all equal contributors, and they titled the paper "Attention Is All You Need." That paper has been cited over 100,000 times, and every one of those eight authors has left Google — several of them founded billion-dollar companies.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 32:08

The architecture they described is the engine running under everything. And the crazy thing is that paper is only 15 pages long, and it's available for free for anyone to read. So the blueprint for the technology powering a trillion-dollar industry is literally just a PDF you can sit down and read right now.

Underneath everything, a transformer is a neural network, but a very specific kind. At the hardware level, it's almost entirely matrix multiplication. The intelligence isn't in any clever logic — it's in the billions of numbers inside those matrices, all of which started random and got tuned during training to produce useful outputs.

Each transformer block has two main operations: attention, where tokens exchange information with each other, and a feed-forward step, where each token gets processed on its own. Both have their own learned weights, and both are trained the same way — by running predictions, measuring error, and nudging every weight in the direction that reduces it.

## Self-Attention

If you zoom in on a block, self-attention is the core operation. For each token, the model generates three vectors from its embedding: a query, a key, and a value. Think of the query as what this token is looking for. The key is what each token offers, and the value is what gets passed along when there's a match. The model compares every query against every key, scores the matches, and mixes the values accordingly. The output is a new vector for each token — the same shape as the input, but now carrying information from the rest of the sentence.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026
https://www.youtube.com/watch?v=YmLp8qe87A0

So the word "bank" in the sentence "the bank of the river was muddy" ends up with a vector shaped mostly by "river" and "muddy." Whereas the word "bank" in the sentence "I went to the bank to deposit money" ends up shaped by "deposit" and "money." Same word in, different vector out — meaning resolved by context.

Multi-head attention runs that operation many times in parallel, with a different set of query, key, and value weights each time. Each run is a head, and nobody tells the head what to focus on. They start random, and during training they end up specializing. One head might learn to track which pronoun refers to which noun, another might track verb tenses, another might track position. It's emergent, not designed.

## Feed-Forward Layers and Stacking

Those outputs get combined back into a single vector per token, and feed-forward layers come next. Each token's vector, now context-enriched from attention, passes through a small two-layer network. The feed-forward step processes each token on its own, refining the signal before it moves on.

Finally, you stack these layers. Attention plus feed-forward is one transformer block, or one layer, and modern LLMs stack dozens or hundreds of these layers. Every block has the same input and output shape — one vector per token — and each layer builds on the last. Early layers tend to capture syntax, middle layers capture meaning, late layers capture reasoning. That's what deep learning means: many stacked layers.

So that's the whole architecture: tokens come in, get embedded, get passed through stacks of attention and feed-forward, and a prediction comes out the other end. Everything else — GPT, Claude, Gemini — is the same recipe, just scaled up.

## Training My Own Transformer

Let's look at how I implemented this to train my own simple transformer model. This next demo is called "train transformer," and the arguments we pass in are the number of epochs, then arguments for the actual generation of the next token — temperature and top-p, which we'll talk about later. Then you have the number of layers in that transformer, and the number of tokens to predict. Transformers only predict one token, but we'll talk about in the next section how we can predict multiple.

We essentially created a transformer that can write simple children's stories. If I run this, it'll actually use a cached model, because I did train this, and training it on my MacBook, with about 14 cores, took over an hour and a half. I'm not training on GPUs, just my CPU, so even training this tiny model took a very long time.

As you can see, our input text is "once upon a time" — that's what we start with — and then we ask the transformer for the next token, and it gave us "a," and then we ask it for another token, and it gave us "small."

If we play around with the temperature and top-p, we'll see it predicts a different next token: "once upon a time, a young." Based on the training data, which I'll show you next, there are different "once upon a

time"s and different characters. So with different values here, it's producing a different next token.

## The Training Data and Probability Distributions

The corpus for this training was a little over 30 short children's stories: "Once upon a time there was a king." "A young prince lived in a castle." "There was once a wicked old man." So we have 30 different short stories, about four or five sentences each, and that's what we actually trained the transformer on. It's learning the relationships between characters, place, and setting, and that's going to allow it to predict the next word when writing out our own stories.

The actual output of our transformer isn't a token — it's a probability distribution. We have our vocabulary, and for every single token in it, the model outputs the probability that any one of those tokens will be the next token given the input prompt. That's all the output is. It's not a single token, it's a probability distribution.

## Sampling: Temperature and Top P

Using temperature and top P, we actually choose which one of those tokens will be the next token. And as we'll talk about in the next section, we basically feed that next token back in, and that's how we can get it to generate more than one token.

## The Transformer Weights File

Now let's take a look at the actual transformer weights file. You can see here we have our vocabulary size — that's what words this transformer learned based on our training data. Then we have our context length, and you can see this is tiny, tiny, tiny compared to the models you talk to on a daily basis. But essentially this transformer has been trained to look at a sequence of 32 tokens and predict what comes next based on those sequences of 32 tokens.

Just like we saw with the embedding model, we actually have the vocabulary stored inside this weights file, as well as the merges, which allow us to tokenize any new incoming input text.

We have a separate embedding model for this transformer, and it has embedding weights as well as positional weights. So the input and output to this transformer is literally the list of every possible token — that's the input, and the output is the list of every possible token. But our prompt isn't every possible token, it's just a few words. So this positional embedding essentially allows the model to learn: okay, the input here isn't everything, it's just these few tokens in this given order. We actually train this positional embedding so that given a user's prompt, it turns it into a list of every possible token, but with the weights tweaked in a way that the model knows what order those tokens are in.

Then we have a list of weights for every single block in the transformer. We talked about how every block has attention, which then goes through feedforward, and that gives us the output — the probability distribution of which token comes next. But we have multiple blocks, so for every single block, we have the weights for query, key, and value, and the weights for the feedforward network for that given block. All of these start off random, just like they do in all of our other models we've trained, but we tweak

them as we train the transformer. Every block has that exact same setup of weights for query, key, and value, plus weights for the feedforward network.

Given that our model has an embedding size of 32, and there are six layers, this model we created actually has 52,000 parameters — that is 52,000 different values that got tweaked. You might contrast that with GPT-OSS 120B, which has 120 billion different parameters in the model itself. So this is peanuts compared to some of these local models, and even those local models are peanuts compared to the models that exist on OpenAI's servers and Anthropic's servers. So this is a 52,000 parameter model that we created here.

## Diving Into the Transformer Code

For each block, it first needs to compute the query, the key, and the value. We do that with matrix multiplication — given those input vectors, we create the query, key, and value matrices, and then we perform the dot product. We do the dot product of query times key, and that gives us the value, which essentially gives us the scores that correspond to the overall attention of any given token to any other token in the input prompt. Those scores give us weights, and that gives us the overall attention values, which we again run through matrix multiplication, giving us the attention matrix for output.

I'm not going to pretend like I understand all the intricacies of this — it's a lot of heavy math — but if you think about the diagrams from the last section, this is roughly how the code corresponds. Ultimately I'm showing it to you because it's not magic, it's just math. It's just numbers, mixing them around, and some really smart people figured out how to mix those numbers around in the right way.

So from this one pass with the query, key, and value, we have the attention matrix. Now we can compute the feedforward based on that attention matrix. We take the input matrix, which is the user's prompt plus positional information, add that to the attention matrix, and then pass that through our feedforward network. That's just simple matrix multiplication to give us our overall output.

The way all of this comes together: what I just showed you happens for one specific block. Given input and positional information, pass that through a block, that gives us an output, and then we can pass that through to the next block. However many blocks we have, it's going to pass through, performing the same steps each time — the input and output size is always the same, we're just passing it from one block onto the next. And then we end up with our final list of probabilities — given these input tokens and positional information, what is the probability for every single token in our vocabulary that it will actually come next, based on the data we trained on.

## From Logits to Probabilities: Softmax

We just saw that the actual output of a transformer is a probability distribution, essentially a score for every token in the vocabulary. Those raw scores are called logits. To actually turn those logits into probabilities, we use an algorithm called softmax. This makes it so that all the probability scores across every single token add up to one.

From there, the model doesn't pick the most probable top token — it samples. As we saw in our example,

there was a temperature variable. A low temperature results in a very predictable output distribution, and a high temperature value results in a more creative, less predictable distribution. For instance, if we pass a temperature of zero, the transformer would predict the exact same token every single time. A higher value, like 1.5, will be much more random and sometimes incoherent.

Our other parameter is top P, essentially a cutoff — only sample tokens covering a certain percentage of the probability distribution. A value of one means give every possible token a chance. A value of 0.2 means only the top 20% of tokens are considered. It essentially shrinks down the possible list of next tokens. Those two parameters are how we get different outputs from the exact same input prompt.

## Auto-Regressive Generation

Everything we've walked through so far — tokenization, embeddings, the transformer layer, softmax sampling — produces one single token. To get the next token, we append the token to the input and then run this entire process again: new attention, new distribution, new predicted token, append, repeat. This whole process is known as auto-regressive generation.

Essentially, the model has no plan. It doesn't know how the sentence ends when it starts. So every coherent paragraph from an LLM emerges one blind step at a time.

## What Actually Gets Sent to the Model

When you send your prompt off to a production LLM like ChatGPT, Claude, or Gemini, it doesn't see just your prompt. It actually sees a structured package that includes the system prompt — hidden instructions written by the AI provider themselves, things like "you're a useful chatbot." It also contains the full conversation history — every single message you've sent in that chat so far. And finally, your latest prompt, your latest message, at the end.

So the model has zero memory between requests. The conversation history is the actual memory of that conversation, and that's what feeds into the auto-regressive loop.

## The Context Window

With that in mind, your prompt, the history, and everything that's gone into your current conversation has to fit into a specific token limit known as the context window. We saw in our simple transformer example that it had a context window of 32 tokens, but modern models have context windows anywhere from 100,000 to millions of tokens. Essentially, the context window is all the transformer can see when it's producing that next token.

That's why, when you have a long conversation, the answers from an LLM might start to degrade, or you might start to see hallucinations — because with a longer context, there are more things the model needs to look at when predicting the next token. It might ignore previous instructions because later messages may have more weight in the overall attention of your chat history.

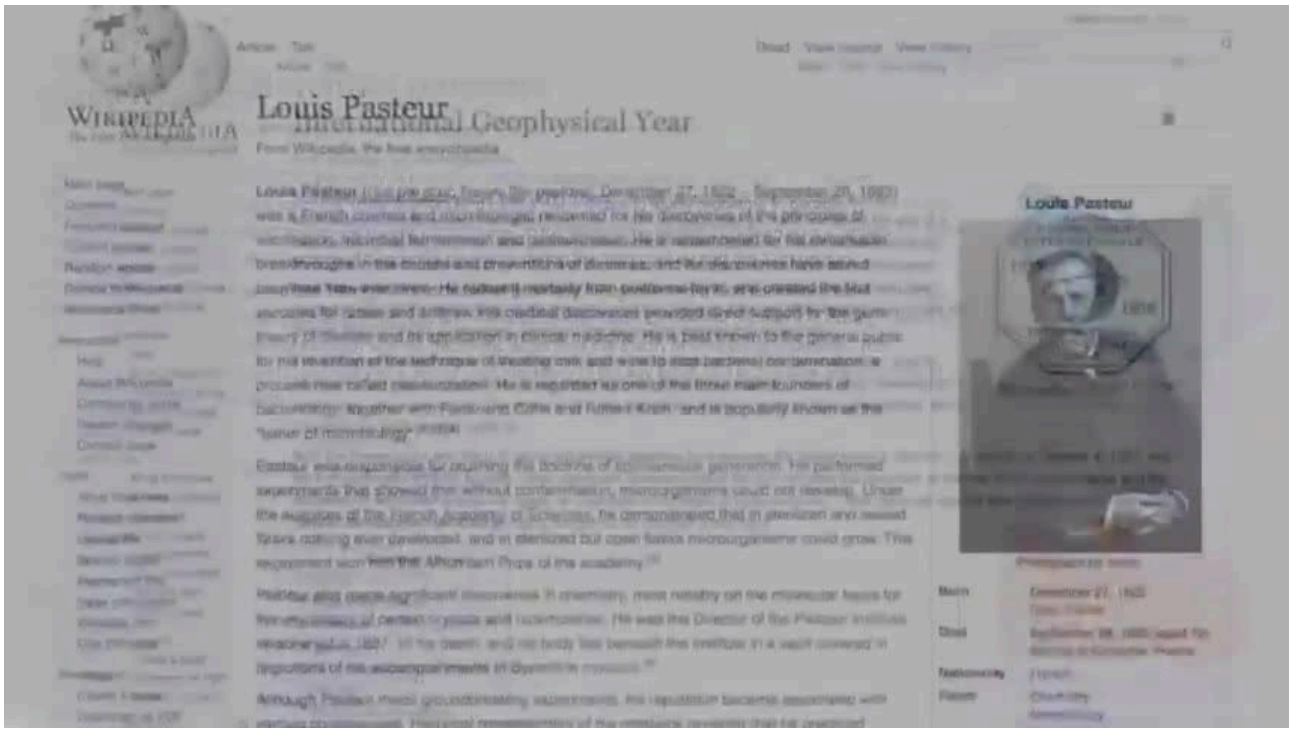
## Putting It All Together

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

At this point, we really have the full picture. Your prompt gets broken down into tokens. Those tokens are defined according to some vocabulary that was predetermined based on the model's training data ahead of time. Then those tokens get turned into embeddings, which actually give the overall prompt meaning. Then the embedded tokens are passed through the transformer: attention gathers context from all of those tokens and passes it through multiple blocks. Finally, a probability score comes out the other end — given all possible tokens in the model's vocabulary, what's the probability that any one of those tokens will be the next token? We then do sampling, using temperature and top-p, to pick that next token. We append that token to the prompt and repeat in a loop.

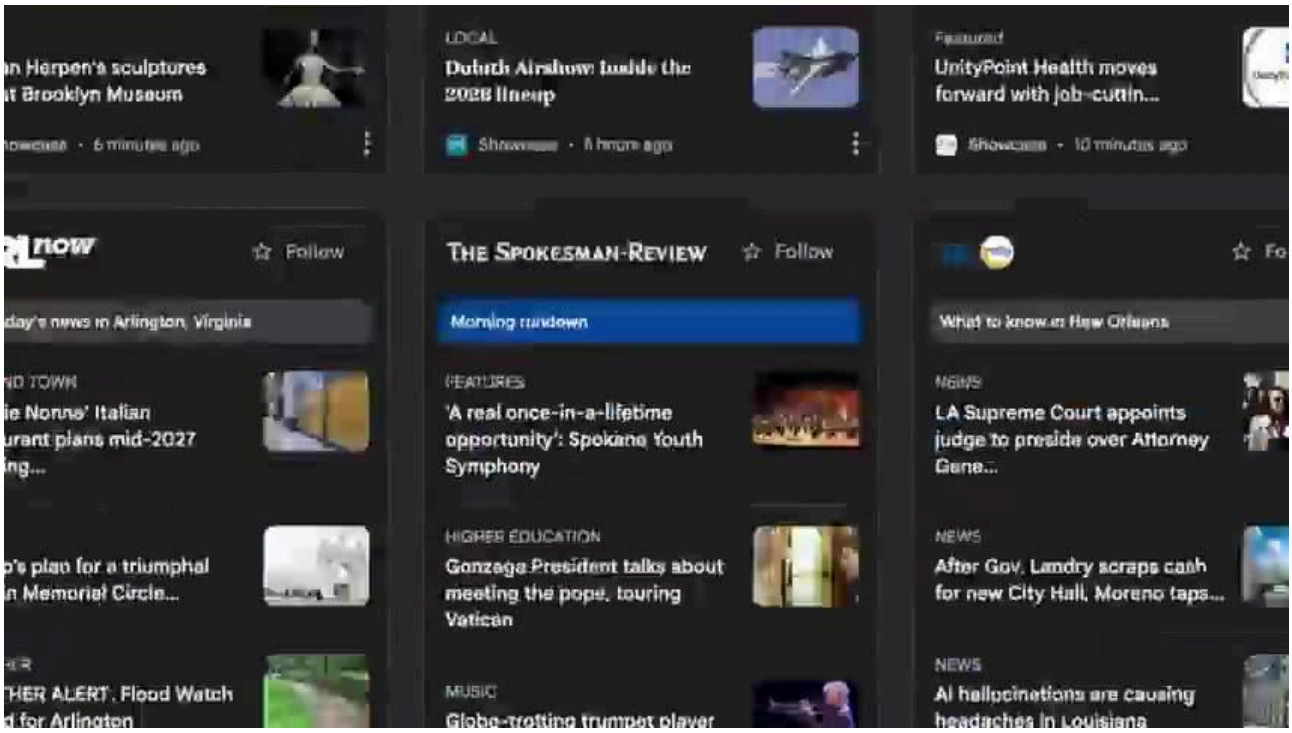



at 45:45

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0



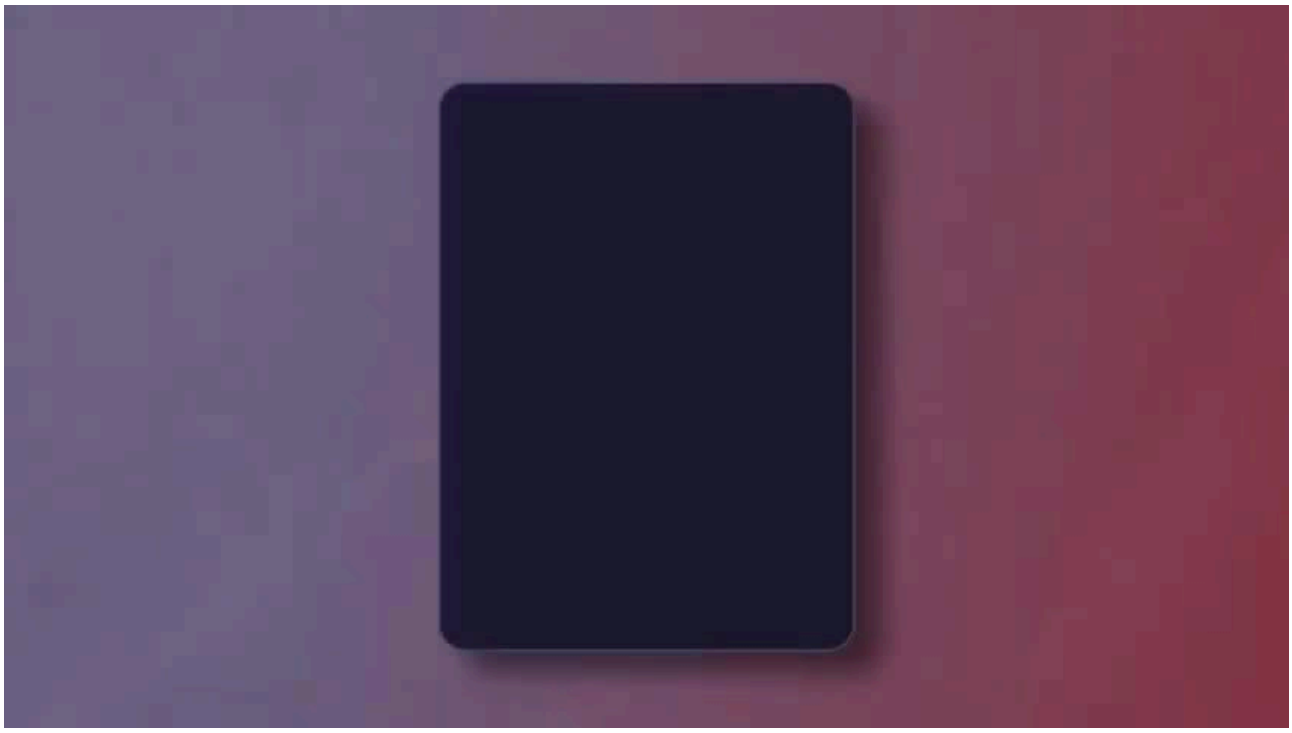

at 45:49


at 45:56

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0



at 46:02

## Pre-Training

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 46:09

Everything we showed in this video is actually just the pre-training phase of a transformer. Essentially, we train a transformer on a massive chunk of the internet — and when I say the internet, I mean it. One of the biggest known datasets is called Common Crawl, a nonprofit that's been archiving the web since 2008 and has petabytes of raw text. On top of that, there's Wikipedia, Reddit, GitHub, digitized books, academic papers, news archives, and a lot of copyrighted material that AI companies won't necessarily admit they actually trained on. But all of this is what transformers like ChatGPT, Claude, and Gemini are trained on.

The result of this training is essentially just a really fancy autocomplete, which is basically what we built in this video.

## Fine-Tuning and RLHF

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 48:37

What really takes this model to the next stage — being more useful as a proficient chatbot — is known as fine-tuning. Fine-tuning is a secondary phase where we take an existing model and pass it through a new set of training data. Our new dataset is a massive, refined list of question-answer pairs in the style of how we'd like the chatbot to actually respond. We then slowly fine-tune the weights within the model so its output gives us more of a chatbot-like response rather than just a fancy autocomplete.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

at 49:19

Then, finally, that fine-tuned model goes through one more step of tuning known as RLHF, or reinforcement learning from human feedback. Real humans are hired to sit down and rate the responses of the model: good chatbot-like responses get ranked high, bad responses get ranked low. What defines "good" and "bad" is entirely up to the company that trains the model, and also up to the judgment of those human rankers.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0


at 49:45

## Tools

The result of all of this is a model that behaves like a chatbot — it answers questions, it's able to have back-and-forth conversations. But it's still just a standalone model. It's a weights file. It just lives on a server somewhere. It doesn't know any more than it's actually been trained on, and it can't reach outside of itself. It can only produce next tokens. This is where the idea of tools comes in.

You can include a list of available tools and how to call them in the context of your request — things like "get the current weather," "search the web for a specific term," "run this code file," or "search the files on the file system." These tool descriptions tell the model what it can perform, and the models are further fine-tuned to output tool calls. So instead of just outputting a response like a helpful chatbot, it will output structured code that says, "Please call this tool." The model itself isn't reaching out over the web or anything like that — it's literally producing a JSON object that says "call this tool."

That's paired with a harness, like a code editor or a desktop app, that can see the model's output was a tool call, and the harness can then actually do what the tool call said to do, like execute code or search the web. The harness takes the result of that tool call and appends it to the conversation history, so the next call to the model has all of that relevant output information in context and can produce a more accurate answer.

There are many other techniques used in modern LLMs to increase the quality of their answers, like mixture of experts and thinking modes, but that's all we're going to cover at a high level in this video. If you're interested in any of those, let me know in the comments and we'll get into it in a future video.

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026
https://www.youtube.com/watch?v=YmLp8qe87A0

## The Dartmouth Workshop, 70 Years Later

This brings us back to the beginning of this seventy-year journey, where in the summer of 1956 four researchers — John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon — gathered about ten people on the top floor of the Dartmouth math department in Hanover, New Hampshire. They had a Rockefeller Foundation grant for $7,500 and a plan for eight weeks of work. They called it the Dartmouth Summer Research Project on Artificial Intelligence, and that was the first time the term "artificial intelligence" was ever used.

They stated: "We propose that a two-month, ten-man study of artificial intelligence be carried out. The study is to proceed on the basis of the conjecture that every aspect of learning, or any other feature of intelligence, can in principle be so precisely described that a machine can be made to simulate it." They thought one summer would be enough.

Now, seventy years later, we're still at it, and a single architecture from a single paper powers products used by billions of people — and we might be one paper away from the next shift. We don't have to be satisfied with the status quo. The transformer might not be the final architecture; it's just the architecture of the current moment, backed by billions of dollars in research funding. Researchers are actively exploring alternatives, and the next breakthrough might look nothing like what we covered today.

## Closing Thoughts

Everything I discussed in this video is based on public research. The papers are public. There are even open-weight models you can download and run yourself. All the code I wrote is linked in the description for you to check out, and there are plenty of other projects on how to build an LLM from scratch you can check out on GitHub.

Personally, as I've done this research and dug more into it, the more I feel like LLMs really are just the most sophisticated pattern-matching autocomplete we've ever built. I don't think they have true understanding. I feel that the less magic there is, the less hand-waving there is in trying to understand the underlying tech, the closer we'll get to really understanding how we can create better methods of working with LLMs — or better understanding of the statistics of language.

I've also thought about how LLMs are trained only on written text, but human intelligence is more than just text. Text is just one form of external communication; it's not actually how the brain itself runs. For me, that reinforces the idea that LLMs are predicting the next token based on language statistics, not necessarily replicating what's happening with human intelligence and what's actually happening in the brain. So, in my opinion, AI is probably better described as something like "alien intelligence" rather than "artificial intelligence" — because it is intelligent, but it's not necessarily a fake version of human intelligence. It's something entirely different.

That's all I've got. Thank you so much for making it to the end of the video. If you have any questions, leave them down below. If I made any mistakes, let me know as well — I'll use the corrections feature of

# I Built an LLM From Scratch

Syntax | 51:48 | transcript saved 24 Aug 2026

https://www.youtube.com/watch?v=YmLp8qe87A0

YouTube and add it so future viewers will see it. And if there's any topic you want to see me dive deeper into or explore in a future video, let me know as well.

All right — see you in the next one.