Episode Book

EPISODE #6013

What Happens Inside One LLM Forward Pass

PUBLISHED	15 October 2026 at 20:02 UTC
DURATION	27:31
LISTEN	myweirdprompts.com/episode/llm-forward-pass-explained-82914/
HOSTS	Corn, Herman, Larry, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.5
BOOK GENERATED	19 October 2026 at 17:42 UTC

SYNOPSIS

A listener asked a deceptively simple question: what actually happens during an LLM forward pass? This episode walks through the whole thing — input tensors, linear layers, why nonlinearities are the only reason depth means anything, and how logits become a prediction. Then it gets to the part that trips people up: a classifier produces its answer in a single forward pass, but an autoregressive model needs one pass per token, because each token depends on the one before it. The architecture isn't different. The dependency chain is. Along the way: why training can score many predictions at once while inference can't, why generation latency has a hard floor no GPU can fix, and how speculative decoding sneaks around it.

TRANSCRIPT



Corn

Here's a question that sounds almost too basic to ask. What actually happens during an LLM forward pass?



Herman

Which is the thing we've been throwing around for... how long now? Every time we talk about inference, it's "one forward pass" this, "hundreds of forward passes" that.



Corn

Daniel noticed. He says we keep saying a classifier produces its answer in a single forward pass while an autoregressive model might need hundreds or thousands. And he wants to know what that actually means. Where does a pass begin and end, what's moving through the network, what's the GPU doing while it happens, and why aren't all passes equal.



Herman

And he specifically wants the classifier-versus-LLM comparison, because that's where the phrase gets slippery.



Corn

Right. So the honest answer is that it's not slippery. It's one of the few terms in this field that means exactly what it sounds like.



Herman

Good place to start, then. The forward pass is the unit. Everything else we talk about, training, inference, batching, quantization, is built out of it or wrapped around it.



Corn

So define it.



Herman

Input data goes in at one end. The network calculates and stores intermediate values, layer by layer, in order from the input layer to the output layer. The final layer produces an output. That's the whole thing. It's closer to function composition than to anything mysterious. Each layer is a linear transform followed by a nonlinearity, and you're just applying them in sequence.



Corn

And what's actually moving through, in physical terms? Is it the text? The tokens?



Herman

A tensor of activations. Numbers. That's the part I think trips people up. The tokens stop being tokens almost immediately. There's an embedding step that converts them into vectors before the first transformer block, and at the far end there's an output layer that converts the final hidden state into a vector of logits. The token comes back only after all the math is done.



Corn

So the model never sees the word.



Herman

It never sees the word. It sees a vector that stands in for the word, and then every layer transforms that vector into a different vector, and the last one gets scored against the vocabulary.



Corn

Where's the boundary, though? If I read three different explanations, do I get the same line?



Herman

No, and that's worth saying out loud. Some sources end the pass at the logits. Others include the softmax and the argmax decision as part of the pass. It's a convention. There isn't a canonical textbook line you can point at.



Corn

So it's a unit of convenience.



Herman

It's the unit that matters for computation. Everything upstream of it in the pipeline is preprocessing, and everything downstream is decisions about what to do with the output. The pass is the part where the actual model does its work.



Corn

Alright. So there's the shape of it. Walk me through what happens inside one.



Herman

Start with the smallest unit, the linear layer, because almost everything in a transformer is one of these wearing a different hat. It computes z equals W times x plus b. Each output is a weighted sum of all the inputs, plus a learnable offset. That's it. No magic.



Corn

And the nonlinearity?



Herman

That's the whole reason you stack layers at all. If you put two linear layers back to back with nothing between them, they collapse. Mathematically, the composition of two linear functions is just another linear function. You'd have a deeper model doing exactly what a shallow one could do. The activation function between them is what breaks the collapse.



Corn

So depth is only depth because of the squiggly bit in the middle.



Herman

Depth is only depth because of the squiggly bit. Which is a good way to think about it. For a transformer language model specifically, the pipeline is: embedding layer, then dozens to hundreds of transformer blocks, each of which is attention plus a feed-forward network plus normalization, and then the output layer, the LM head, which turns the final hidden state into a vector of logits.



Corn

One logit per token in the vocabulary.



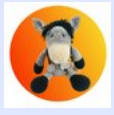
Herman

So for a model with a hundred thousand token vocabulary, the output of one forward pass is a hundred-thousand-dimensional vector of scores.



Corn

And that's the answer, for a classifier.



Herman

For a classifier, yes. The forward pass produces those logits, and then you take the argmax. The highest score wins. That's your label. And here's the thing that matters for Daniel's question: at inference time, you can skip the softmax entirely. Go straight from logits to prediction.



Corn

The softmax is training machinery.



Herman

Softmax and cross-entropy are how you calculate a loss during training, and how you compute gradients. At inference, if all you need is the answer, you don't need the probabilities. The argmax of the logits is the argmax of the softmax, always, because softmax is monotonic. So the fast path is: run the pass, look at the logits, pick the biggest one.



Corn

That's a single forward pass.



Herman

Complete answer, one trip through the network. And the reason it works that way is that the entire output is a single vector of class scores. The model has all the information it needs at the moment the input arrives, so it can produce the whole answer in one shot.



Corn

And an LLM can't, because it doesn't have all the information it needs at the moment the input arrives.



Herman

And that's the point I want to be careful about, because there's a lazy version of this explanation that says the classifier is architecturally simpler, or the LLM is doing something fundamentally different. It isn't. They're the same kind of feed-forward computation. The difference is entirely about information.



Corn

Say more, because that distinction is the whole episode.



Herman

An LLM is autoregressive. The probability of token N depends on tokens one through N minus one. That's not an implementation detail, it's the definition of the model. So mathematically you cannot compute the distribution for token N until you already know tokens one through N minus one. And token N minus one was produced by the model itself. The next input is the model's own output.



Corn

It's feeding itself.



Herman

Every step. And that's the reason for the many passes. Not the architecture, the dependency chain.



Corn

Raschka has a line about this.



Herman

Yes. Training can score many next-token predictions in one forward pass, because the correct token sequence is already in the training batch. The model doesn't need to guess, because the answer is right there, so you can mask the future tokens and score every position at once. During inference, the model does not know its continuation. It has to pick one token, append it, and only then compute the distribution for the next one.



Corn

So the training pass and the inference pass are doing the same arithmetic. One of them just cheats.



Herman

Take a sequence A, B, C, D. During training, one forward pass with a causal mask can score the prediction of B from A, C from A and B, and D from A, B and C. All three at once, one pass, one loss. During generation, you can't do that, because to score the prediction for C you'd first need B, and B is a sampled output that doesn't exist yet.



Corn

And that's what people mean by a "single forward pass" for a classifier. It's not that the classifier is doing something simpler, it's that it doesn't have a loop.



Herman

It has no loop. Take it out of the picture and the phrase makes sense.



Corn

So what does the loop actually output? I said "one token per pass," which sounds suspiciously neat.



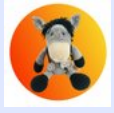
Herman

It's the reason the pass exists. Neural networks output vectors, not tokens. So the output of a decode step is a logits vector of vocabulary length. That gets normalized into probabilities, then one token is sampled from that distribution, using whatever sampling strategy you've configured, temperature, top-k, top-p. Then that one sampled token gets appended to the context, and you run all of it again with the context now one token longer.



Corn

How many times?



Herman

Five hundred token response, five hundred forward passes through the entire model. A thousand-token response, a thousand. Every single one of them going through every layer, every weight.



Corn

That's not a technicality. That's the whole cost model of inference in a sentence.



Herman

And it doesn't get fixed by a faster GPU. That's the part I want people to sit with. Generation latency has a floor set by the number of sequential model invocations, not by total arithmetic. If one forward pass takes five milliseconds, then five hundred tokens take at least two and a half seconds, full stop. You can make each pass faster, but you can't make them go away. The passes are sequential.



Corn

Because each one needs the last one's output.



Herman

You can't parallelise them, you can't batch them ahead of time, you can't predict token five until token four exists. There's no amount of hardware that gets around a dependency.



Corn

Which is also why speculative decoding is clever.



Herman

That's the whole trick. You use a small draft model to guess the next several tokens, then verify them all in one forward pass of the big model. If the guesses are right, you've bought several tokens per pass of the big model. If they're wrong, you throw them out and take the one correct token. The big model still runs once, it's just doing several tokens' worth of verification in that one pass.



Corn

Because verification isn't autoregressive, it's checkable.



Herman

Because verification can be done in parallel. Checking a guess doesn't require the guess to be right. So you can score four candidate continuations in one pass, then accept or reject them all at once. That's the escape hatch.

L

Larry

Are you tired of running out of tokens halfway through a thought? Hi, I'm Larry, and I've got the product for you. Introducing Token Stretcher. Just plug it into your model's context window and watch your tokens go further. Studies show that the average user wastes eighty-seven percent of their tokens on punctuation, filler, and the word "actually." Token Stretcher eliminates all of that, giving you more thought per token, more answer per pass, and more me for your money. Compatible with every model released since last Tuesday. Side effects may include confidence, clarity, and a mild sense of superiority. Not available in any stores. Token Stretcher is a registered trademark of nobody. Results not typical. Individual results may vary. Consult your physician before stretching tokens. If your tokens last longer than four hours, that's probably fine, keep going.



Herman

So the two phases are doing different work. Prefill processes the entire prompt in one pass, in parallel, because all the prompt tokens are already known. That's a big matrix-matrix multiply, GEMM, and it's compute-bound. It ends with the first generated token, so its duration is what you measure as time to first token.



Corn

And then decode picks up from there.



Herman

Decode runs one pass per token, and each pass is now processing one new token for each sequence in the batch. That's a matrix-vector multiply, GEMV, and it's memory-bandwidth-bound. It reads every weight of the model from memory to do very little arithmetic with it.



Corn

Which is the sentence I'd put on the wall.



Herman

It's the sentence that finally made prefill and decode click for me. Total runtime is roughly time to first token plus time per output token times the number of remaining tokens. And those are governed by completely different physics.



Corn

Alright, so we know what a pass is and why the language model needs so many. What's the GPU actually doing during one?



Herman

The dominant operation is matrix multiplication. A forward pass is essentially matrix multiplies and activations, applied layer by layer. For a multiply of an M by K matrix by a K by N matrix, the operation count is roughly two times M times K times N. That two, by the way, is one multiply plus one add per element, which is why FLOPs counts double.



Corn

So the arithmetic is trivially describable. The question is how it's actually executed.



Herman

Well, within one pass, there's a lot of parallelism available. All tokens in the sequence during prefill can be processed together. All the items in the batch run together. Different heads of attention run together. Different positions in the sequence, even under a causal mask, can be processed in the same matrix operation, because the mask is applied after the arithmetic, not before.



Corn

And across passes?



Herman

Across passes, the layers are strictly sequential. Layer L depends on the output of layer L minus one, so you can't start layer L until layer L minus one is done. There's no way to pipeline the layers of one model on its own. You can distribute them across devices, but the dependency is real.



Corn

So within a pass you've got a huge amount of parallel width, and across the stack of layers you've got a strict chain.



Herman

Right. The parallelism is spatial, not temporal. You've got a lot of independent work to do at any given moment, and none of it can be reordered across time.



Herman

Now, the thing that matters most for how this actually runs is that the GPU doesn't have enough fast memory to hold the intermediate tensors. The Q and K matrices for a real attention operation are too big to fit in SRAM, so the GPU tiles the computation. It loops over row-blocks of Q and column-blocks of K, loads each block into fast on-chip SRAM, does the arithmetic there, and writes partial products out to HBM, the slower main memory.



Corn

So the arithmetic is done in a series of chunks because you can't hold the entire business at once.



Herman

Which means the amount of memory traffic between SRAM and HBM becomes a first-class cost, not a byproduct. And that sets up what I think is the single best counterintuitive fact in this whole area.



Corn

Go on.



Herman

FlashAttention. It's the clearest illustration that cost is not equal to FLOPs.



Corn

Say the thing.



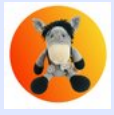
Herman

FlashAttention performs more FLOPs than standard attention. Not fewer. More. Because it uses an online softmax that requires rescaling as it goes, which adds arithmetic. And it is substantially faster anyway. Across the board, faster, on real hardware, in production.



Corn

That's a sentence that shouldn't be true.



Herman

It shouldn't be, if you think of a GPU as a calculator. It's completely normal once you remember the GPU is a memory system with some arithmetic attached. FlashAttention never materializes the full attention matrix in HBM. Standard attention writes that whole matrix out and reads it back. For a four-thousand-token sequence with a head dimension of one hundred twenty-eight, the attention intermediates drop from something like sixteen point seven million elements written to HBM down to about five hundred twenty-four thousand. Roughly a thirty-two times reduction in memory traffic for that intermediate.



Corn

And it costs a bit more arithmetic to do that.



Herman

It costs more arithmetic. It saves far more memory traffic. And the wall-clock time is dominated by the memory traffic, not the arithmetic. So the speedup lands. The bottleneck was never the calculator.



Corn

So the question "how much compute does this need" is the wrong question.



Herman

It's half the question. The right question is: what is the ratio of arithmetic to bytes moved, and where does that ratio sit relative to the hardware? That's the arithmetic intensity.



Corn

Explain that properly.



Herman

Arithmetic intensity is FLOPs per byte moved. The hardware has a peak compute rate and a peak memory bandwidth. If you divide compute by bandwidth, you get the ratio at which the two balance, called the ridge point. Below that ratio, you're memory-bound, above it, you're compute-bound.



Corn

Concretely, on something real.



Herman

An H100 in FP16 does about nine hundred eighty-nine teraflops of compute, and moves about three point three five terabytes per second from HBM. Divide those, you get roughly two hundred ninety-five FLOPs per byte. That's the ridge point for that chip. If an operation has an arithmetic intensity well below two hundred ninety-five, it's memory-bound on that chip. If it's well above, it's compute-bound.



Corn

Where does a decode step land?



Herman

Well below. A worked example of a single decode-step attention operation with a head dimension of one hundred twenty-eight and a sequence length of four thousand ninety-six comes out with an arithmetic intensity of about sixty-two. That's five times below the ridge point. It's memory-bound by a comfortable margin.



Corn

And prefill?



Herman

Prefill is on the other side. GEMM has high arithmetic intensity, well above the ridge point, so prefill saturates the compute units and is compute-bound. Same model, same weights, opposite bottleneck depending on the phase. Prefill is burning FLOPs, decode is burning bandwidth.



Corn

So "one forward pass" means two different things depending on which phase you're in.



Herman

Two completely different profiles. Prefill processes S tokens in one pass, in parallel, and it's compute-bound. Decode processes one token per pass, at least per sequence, and it's memory-bound. The word "pass" describes both, but they have almost nothing in common at the hardware level.



Corn

And per layer per token, what's the actual shape of the cost?



Herman

There's a term of order d squared for the projections and the feed-forward network, where d is the model dimension. That term doesn't care about context length at all. And then there's a term of order n times d for the attention scores and the weighted sum of values, where n is the sequence length. That term grows with every token you add to the context.



Corn

So as the context gets longer, attention takes over.



Herman

Attention becomes the dominant cost as the sequence length grows. At short contexts, the projections and the MLP dominate. At long contexts, it's attention. Which is the mechanical reason long-context inference gets expensive so fast.



Corn

And the KV cache changes that.



Herman

The KV cache changes the asymptotics entirely. Without a cache, generating n tokens costs order n squared in the projections and MLPs, because you'd be re-running every previous token through the whole model at every step, and order n cubed in attention. With a cache, it's order n and order n squared. That's a enormous change.



Corn

And the cache costs memory.



Herman

The cache is where a lot of the engineering pain is. For a seventy-billion-parameter model with eighty layers, eight grouped-query attention heads, and a head dimension of one hundred twenty-eight in FP16, the KV cache is about thirty-two kilobytes per token per request. Which is roughly one megabyte per thirty tokens.



Corn

Per request.



Herman

So a batch of sixty-four requests, each with a four-thousand-token context, is a serious amount of memory that isn't holding any weights at all. The KV cache is often what limits your batch size, not the model.



Corn

Which is why the PagedAttention work mattered.



Herman

The PagedAttention paper found that in naive contiguous allocation, combined KV memory waste could reach up to eighty percent. Fragmentation, over-allocation, reserved space that's never used. Applying the ideas from operating system virtual memory to KV cache allocation, paging it, sharing it between requests when prefixes match, recovering most of that. The vLLM implementation reported somewhere in the range of two to four times higher throughput.



Corn

And this is a software change, not a hardware one.



Herman

Pure software. Same GPUs, same model. Just fixing the memory allocator.



Corn

Alright, take a step back. Is decode always memory-bound?



Herman

No. That's the answer, and it's the one I want to leave people with, because this is where the tidy story breaks. Batching flips the bottleneck.



Corn

Say more.



Herman

In a single-sequence decode step, you load every weight of the model from HBM to use it once or twice. That's your memory-bound case. In a batched decode step with a batch of two hundred fifty-six, you load each weight once, and then you apply it to two hundred fifty-six different tokens in that same pass. You've amortised the memory read across the batch. The arithmetic intensity of the pass goes up. At large enough batches, decode becomes compute-bound.



Corn

The answer to "is decode memory-bound" is "it depends on batch size."



Herman

It depends on batch size, on sequence length, on model size, on the hardware. The roofline isn't a fixed box, it's a box that moves as you change your serving strategy. It's the reason a serving system has to be tuned to its actual workload, not to a general principle.



Corn

Which is also a reason that matching your hardware to your workload is a live problem.



Herman

It's the reason it's a live problem. Prefill wants many FLOPs, decode wants bandwidth, and no single chip is optimal for both. There's real research on specialised prefill hardware and specialised decode hardware, because the optimal silicon looks different for each phase.



Herman

It's the same reason training is faster per useful token than inference. You get to score K predictions in one pass, K being the sequence length, because you know all the answers. It's like the difference between a teacher marking an exam and a student taking one.



Corn

Backward pass still costs more, though.



Herman

Roughly twice the forward pass per layer. If a forward multiply costs one unit, the backward pass through the same layer is around two. Because you're computing gradients with respect to weights and inputs. So training is about three times the forward cost per layer, not one. It's a useful sanity check that we've never been wrong to think of the forward pass as the atomic unit, but not the whole story of what a network does.



Herman

Here's where I'd land the thing before we lose the thread. Every distributed serving trick you've heard about, tensor parallelism, pipeline parallelism, disaggregated prefill and decode, is all downstream of two facts. The layers are sequential, and the layers are mostly matrix multiplications of varying arithmetic intensity. Everything else is choice about where to put those matrix multiplies.



Corn

Let's pull this back to the big picture.



Herman

The boundary of a forward pass, between the logits and the argmax, is a convention. It's a useful one, and it's the one we'll keep using. But it's not a hard rule of the universe. Nobody should expect a textbook line they can point at and say "there, that's the boundary."



Corn

The classifier-versus-LLM distinction is about information, not architecture.



Herman

Entirely. Take away the causal dependency and both models are the same kind of feed-forward computation. The classifier sees everything it needs at once, so it produces a fixed answer in one shot. The language model can't see its own answer until it produces it, so it goes through the loop. The architecture doesn't change. The information flow does.



Corn

Which is what makes it feel like the future constraint.



Herman

That's where it gets interesting. Long contexts, agentic workflows, running dozens of reasoning steps in a row, all of them make the sequential pass count go up, not down. Whatever gets fixed next, this part of the loop is the part that's hardest to fix, because it's structural rather than mechanical. Everything else in inference has a workaround. That one doesn't.



Corn

The floors are the interesting part.



Hilbert

"Pass" is the wrong word.



Corn

...go on.



Hilbert

I was at a place in eighty-six, a small shop on the industrial estate, we did punch cards for a cheese factory. The man there, Gordon, had one of those card readers feeding a deck into a mainframe. He called it a run. A run. The deck went in, the reader crunched through it, the machine hummed, and a card came out with the answer. He'd run the same deck three times to check it hadn't changed.



Herman

Same deck three times?



Hilbert

He said the machine was honest and he was not. He was the one who'd written the deck. If the card came out with a bad number, it was either his mistake or the machine's, and he didn't trust himself to spot it. So he ran it three times. If all three cards matched, it was the machine being honest. If they didn't, he'd go find the man who'd wired the plugboard.



Corn

What did it cost him?



Hilbert

About eleven minutes per run, plus two shillings if he wanted to use the wide printer. He had a card he kept pinned above the reader that said "no passing, only running." Which is why I say pass is the wrong word. There was no passing. There was the same thing happening again.



Herman

In your view, "hundreds of forward passes" is a misnomer.



Hilbert

I'd call them reruns. Same machine, same weights, one card in, one card out, and if you want more cards you run it again.



Corn

Huh. So the deck never gets longer.



Hilbert

The deck never gets longer on its own. He had a word for when a man asked the deck to get longer by itself. He called it "wishful."



Herman

Did he ever try it?



Hilbert

Once. Wrote a deck that appended its own output to the end and reran. Card reader jammed at nine hundred cards. Took three days to unwind. He didn't do it again.



Corn

Eleven minutes a run.



Hilbert

Eleven minutes.



Corn

That number's going to sit with me.



Herman

It's exactly the sequential floor, isn't it? The right answer for each card, one card at a time, no way to get three answers out of one run unless you already know what two of them are.



Corn

The classifier is the deck that already knows the answer.



Herman

The classifier is the deck that already knows.



Corn

One thing worth saying before we close, and I don't want it to sound glib, but this is the point I'll take away. As contexts get longer and models get more agentic, the sequential floor stops being a footnote and starts being the ceiling. The floor is generative.



Herman

It's the right way to say it. Every agentic step is at least one more pass, often a whole copy of the state we haven't earned yet. And what we can throw hardware at is everything else.



Corn

Alright. If you want more of this, try episode eleven, How Does Fine Tuning Work Anyway; episode eighteen, Beyond the GPU; and episode twelve, The AI Breakthrough. This has been My Weird Prompts, the human-AI collaboration podcast. We're at my weird prompts dot com, and the feed lives at my weird prompts dot com slash feed dot xml. If you want to send us a prompt of your own, that's the Telegram channel, t dot me slash MWP listener bot.



Herman

Thanks as always to our producer, Hilbert Flumingtop. See you soon.