Looped Transformers: Thinking twice with the same weights

How Inkling 2 Mini and GLM-5.4 run their layers several times, and what tiny transformers trained from scratch can teach us about it

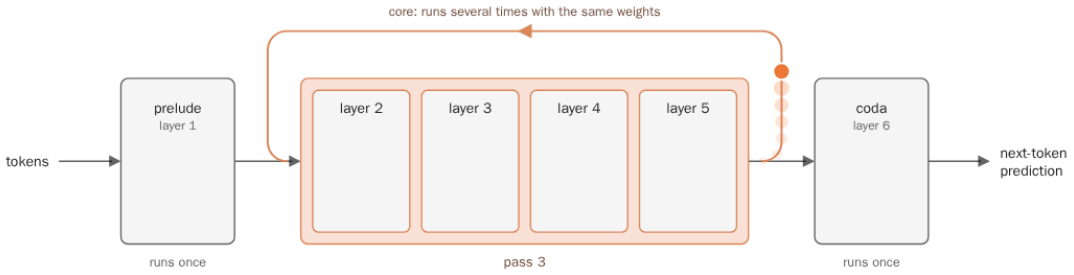


Table of Contents

1 Inkling 2 Mini and GLM 5.4

- 1.1 Depth and size are two different things
- 1.2 Inkling 2 Mini: designed to loop
- 1.3 GLM-5.4: a large model converted into a looped one

2 Understanding looped transformer with small experiments

- 2.1 A treasure hunt for tiny transformers
- 2.2 Following a chain: stored layers or applied layers?
- 2.3 Memorising facts: where parameters win
- 2.4 Choosing the number of passes at test time
- 2.5 Watching the loop work

2.6 Keeping the loop stable

2.7 Conclusion: What the tiny models say about the big ones

2.8 Reproducing the experiments

2.9 Sources

Last week, Thinking Machines and Z.ai both released new language models that do something that might first sound like a bug: for every token they process, they run part of the network *several times in a row*, with exactly the same weights.

Thinking Machines' Inkling 2 Mini runs its 12 middle layers three times, while Z.ai's GLM-5.4 runs 28 of its layers twice. Re-using the same layers several times does not change the parameter count of the model, so *why would running the same layers again make a model any smarter?* What does a second pass compute that the first one could not? And what does it cost?

In this post I would like to do two things.

First, I'll explain how these looped transformers work, using what we know about the models that the two labs have published.

Then we'll try to understand what this looping technique brings, and for that I'll train a few very small looped transformers myself to see the principle at work (⋯tiny models actually: about a million parameters each, a few minutes to train on a regular GPU).

1 Inkling 2 Mini and GLM 5.4

1.1 Depth and size are two different things

Let's start with an ordinary transformer. During processing, each token is represented by a vector, often called the *residual stream*. A layer reads this vector (and of course, through attention, the vectors of the other tokens), computes an update, and adds it back to the stream. The next layer reads the result, adds its own update, and so on. A model with 40 layers applies 40 such updates, one after the other. (If you count the attention block and the MLP block separately, that's 80 updates to the residual stream.)

Two quantities are hiding here, and in a vanilla transformer model they are tied together:

- the size, meaning how many parameters the model stores;
- the depth, meaning how many sequential steps of computation each token goes through.

For a regular transformer, each new layer adds one step *and* one set of weights. It means that if you want a deeper model, you have to store a bigger one, with more parameters.

A looped transformer unties these two concepts. You take a block of layers and apply it several times, feeding its output back into its input. Three passes through a 12-layer block give 36 steps of computation for the price of 12 layers of weights.

In practice, you don't want to loop the whole network. The usual layout has three parts:

- a prelude: a few layers that run once and turn token embeddings into the model's internal representation;
- a core: the block that actually gets repeated;
- a coda: a few layers that run once at the end and turn the internal representation back into a prediction for the next token.

A standard transformer and a looped one

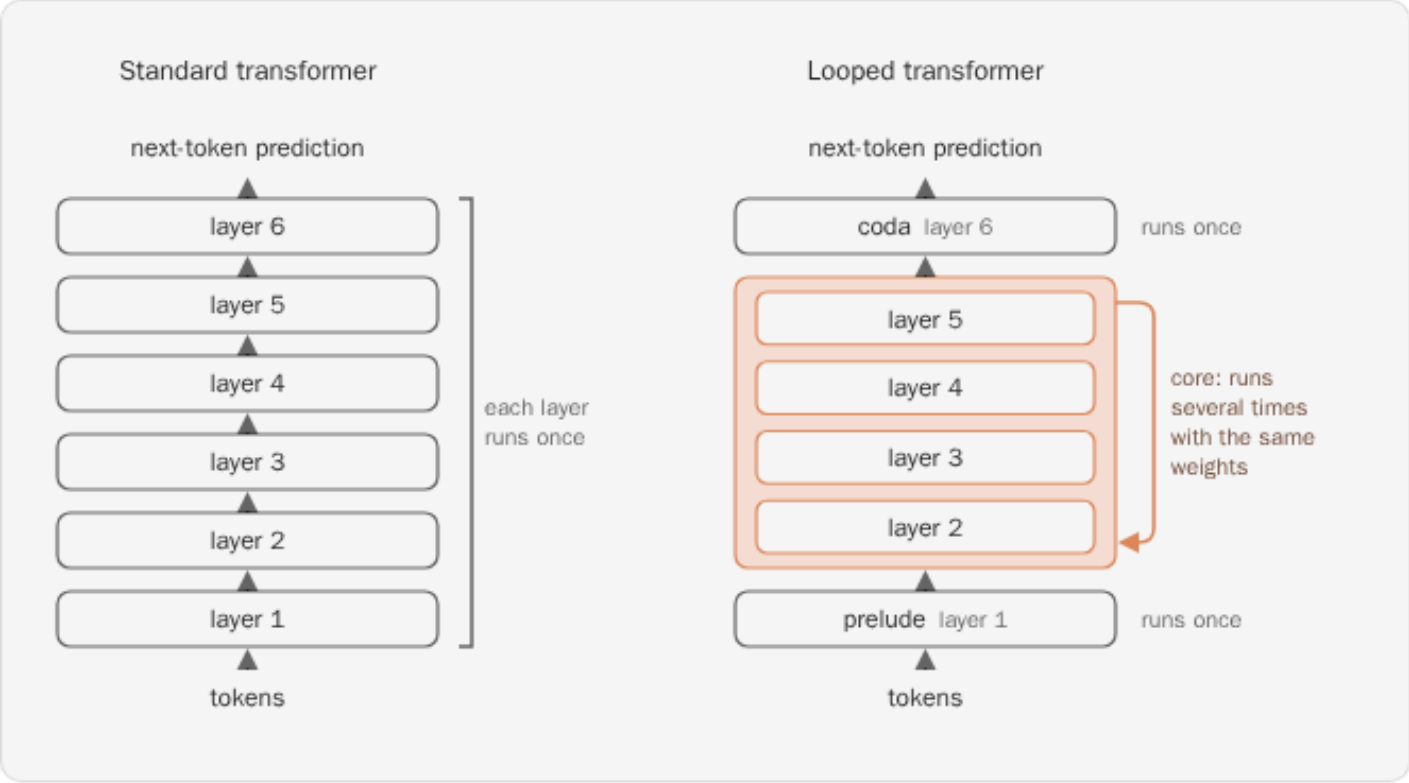


Figure 1 · In a standard transformer, each layer has its own weights and runs once. In the looped version of the same six layers, the prelude (layer 1) and the coda (layer 6) run once, while the core (layers 2 to 5) runs several times with the same weights, its output fed back into its input.

The intuition behind this layout (stated in Thinking Machines' release) is that the first and last layers do translation work between tokens and internal concepts, while the middle layers do the more abstract processing. Translation needs to happen only once. The abstract processing might benefit from more rounds.

Reusing layers is not new (see side note), so why the sudden wave of releases? A possible reading is that two things are pushing in this direction. First, labs want models that reason more, and reasoning seems to need sequential steps more than it needs storage. Chain-of-thought is a way to do it, but looping is another way to extend test-time compute, as we will show in one of our experiments later. Second, models increasingly run where memory is scarce and compute is relatively cheap: a laptop, a single GPU, a phone. With RAM being scarce, every stored parameter eats some memory, while an extra pass through weights already sitting in memory costs comparatively little (that being said, GLM-5.4 is not meant to be run locally!).

The *Universal Transformer* (Dehghani et al., 2018) already applied the same layer over and over, with a mechanism to decide when to stop. *ALBERT* (Lan et al., 2019) shared weights across all its layers, mainly to make BERT smaller. Theory papers then showed that a looped transformer can, in principle, emulate a programmable computer (Giannou et al., 2023), and that a few layers looped many times can match much deeper models on reasoning tasks (Saunshi et al., 2025). In 2025, *Huginn* (Geiping et al., 2025) introduced the prelude/core/coda vocabulary in a 3.5B-parameter model whose number of loops can be chosen at test time, and ByteDance's *Ouro* models (Zhu et al., 2025) pushed looped pretraining to several trillion tokens.

1.2 Inkling 2 Mini: designed to loop

Inkling 2 Mini has 24 layers, grouped into four blocks of six. The first block (the prelude) runs once, the two middle blocks (the core, 12 layers) run three times while the last block (the coda) runs once. That makes 48 layer applications per token, from 24 layers of weights.

It is also a MoE model: 26.4B parameters in total, with each token routed to 8 of 128 experts in each layer. One pass activates 3.4B parameters, and each extra pass through the core adds about 1B. At three passes the model therefore does the work of a 5.4B-active model while storing the weights of a 26B one. The checkpoint weighs about 20 GB, small enough for a 32 GB laptop.

Three details of the loop are worth noting:

- Every pass restarts from the prelude's output. Passes 2 and 3 also receive a "message" from the previous pass, merged in through a learned gate.
- That gate starts fully closed. At the beginning, each pass behaves exactly like a non-looped model, and the gate opens progressively during training. Thinking Machines says this let them introduce looping partway through training without destabilising the model.
- Each of the 48 layer applications keeps its own key-value cache, so *the memory cost of attention still grows with the number of passes*.

The release also includes a few useful measurements. Looping the middle 12 layers beat looping the last 12 on 27 out of 28 evaluations. And when they compared it with a non-looped model from the same series, the non-looped one cost 7.5% less compute to train, but needed 140% more *inference* compute to reach the looped model's loss (measured on long agentic traces).

Finally, Thinking Machines released a research sibling, TML-d24-varloop-research, trained with 1 to 5 passes, so that you can choose the number of passes when you load it. Its model card reports how scores change with the number of passes. I'll come back to these numbers at the end, and we will see the same thing at work with our small models.

1.3 GLM-5.4: A LARGE MODEL CONVERTED INTO A LOOPED ONE

GLM-5.4 is a different beast. It is a ~750B-parameter model, and Z.ai did not train it as a looped model from scratch. They took the already pretrained GLM-5 base and converted it, through continued training, into a model that runs 28 of its layers twice. Their justification is *"looped models trained from scratch still lag behind unshared models under the same pre-training compute budget."*

That being said, converting a model that was never meant to loop raises a practical problem. If the second pass simply overwrites the hidden states, Z.ai writes, *"activation norms blow up across iterations"*, and the layers downstream receive vectors of a size and distribution they have never seen. Their fix is a gate per token and per pass, which decides what fraction of each pass's update gets written back. After training, the first pass writes cautiously (small gate values) and the second pass opens the gate wide. Harder tokens get larger first-pass updates.

They made two other changes that help to keep the cost down. The second pass reuses the attention cache selected by the first pass and keeps only a short sliding window of its own, so *"an extra loop only costs one sliding window of KV cache rather than a full context cache"*. And it also reuses the first pass's sparse-attention indices instead of recomputing them.

Z.ai's evidence comes from a controlled test on a 30B model: at equal parameter count, they found the looped version cost about 10% more training compute and improved results, mostly in math, scientific reasoning and long-context tasks.

Here is a quick comparison of the two models:

	Inkling 2 Mini	GLM-5.4
Total size	26.4B (MoE)	~750B
Looped part	middle 12 of 24 layers	28 layers of the GLM-5 base
Passes	3 (1 to 5 in the research variant)	2
Connection between passes	restart from the prelude + gated “message” from the previous pass	per-token, per-pass gate on the update
Training	looping introduced during training	converted from a trained non-looped model
Attention cache	one per layer application	second pass reuses the first pass’s cache + a sliding window

2 Understanding looped transformer with small experiments

2.1 A treasure hunt for tiny transformers

To highlight and quantify the benefits of looped architectures, these labs can afford to compare looped and non-looped models with billions of parameters⋯ but I can’t! But the claims above are about *principles* (depth versus storage, reasoning versus knowledge), and these principles can be tested on a small scale if you pick a much simpler task.

So here is the one I decided to play with. *Think of a treasure hunt where every clue tells you where the next clue is hidden* and your goal is to follow them to the last one. It is like following the edges on a directed graph until you can’t go further.

Concretely, in my synthetic task, there are 64 symbols (call them nodes). The input is a set of arrows `a → b` that link the nodes into several separate chains (so they represent edges in the graph). In my artificial dataset, for a given input, *the arrows come in random order*, and the model has no positional information at all, so the input really is an unordered set. A last token holds the *query*, one of the nodes. The model must output the last node of the query’s chain.

If that’s a bit abstract, here is an example:

```
arrows:  17→42   5→9     42→3    9→61    3→28    61→12   6→8
         23→31   43→41   58→1    45→48   11→19   48→51   46→35
         40→22   60→16   57→32   54→11   20→4    14→38   26→34
         53→54   2→46    35→59   24→44   56→10   49→27   55→57
         39→63   0→37    37→40   50→7    29→49   38→33   18→15
         41→20   15→2    63→36   59→55   44→58   8→14    25→21
         31→43   47→29   34→0    12→30   1→6     32→13   19→45
query:   17
```

Here the chain to follow is `17 → 42 → 3 → 28`, then `28` has no outgoing arrow so the token to predict is `28`. And notice that to get there we had to perform 3 hops. This is illustrated below.

The task and the model

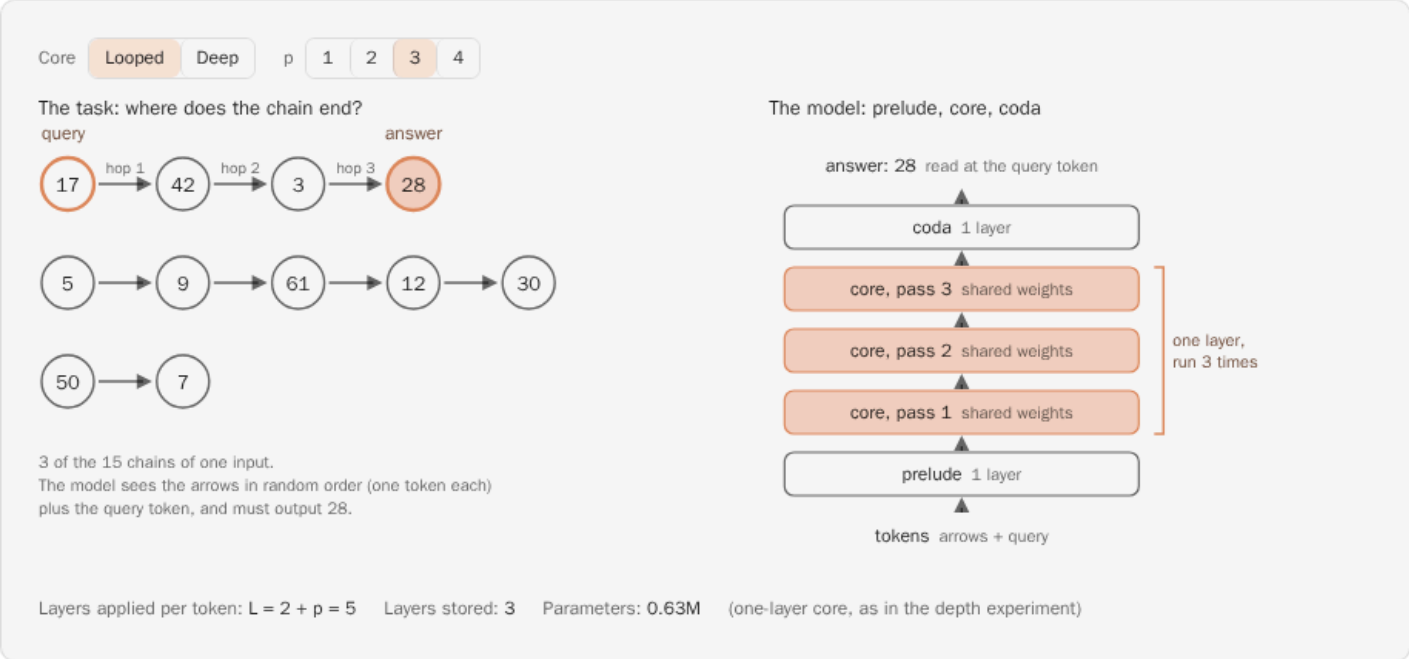


Figure 2 · Left: three of the chains of the example above. Right: the model. Switch between a looped core and a deep one, and change *p*, to see how many layers the model applies and how many it stores.

The figure shows only three chains. A real input uses all 64 nodes, which in this example form 15 chains. Here is the complete input, exactly as the model receives it: 49 arrows in random order, each one a single token, followed by the query token.

What I like about this task is that the number of hops that have to be done to answer correctly is a dial: I can set exactly how many reasoning steps a question requires. And nothing has to be memorised, since everything the model needs is in the input. In a way it is a stripped-down version of something language models do all the time: following a variable through a piece of code, or chaining "the manager of the manager of Alice". It is also close to a benchmark on which Inkling's variable-loop model improves a lot with more passes (GraphWalks, which asks models to walk through a graph given in the context).

All models below share the same ingredients: 128-dimensional vectors, 4 attention heads, one prelude layer, one coda layer, and a core in between that is either looped or not. They have between 0.6 and 2 million parameters and each one trains in 3 to 30 minutes on a single A10G GPU (I ran on Hugging Face Jobs). During training, the maximum chain length grows progressively (a *curriculum*), because I found in a first pilot that models trained directly on long chains learn nothing at all.

2.2 Following a chain: stored layers or applied layers?

So here is the first question we want to answer: *When a model follows a chain, what matters: how many layers it stores, or how many it applies?*

To find out, I trained two families of models on *chains of up to 48 hops*. In the looped family, the core is a single layer, repeated 1 to 4 times. In the deep family, the core contains 1 to 4 distinct layers, each run once. Counting the prelude and the coda, both families apply between 3 and 6 layers per token (I'll call this number L). The difference is in what they store, because the looped models always have 0.63 million parameters, while the deep ones grow from 0.63 to 1.22 million.

Before the results, we could try a prediction. The naive strategy is of course to follow one arrow per layer, so 6 layers would get you 6 hops. But attention allows something much better, because there is a classic trick from parallel computing called *pointer jumping*. Suppose every arrow `a → b` asks, all at the same time, "where does `b` point?". After one round, every node knows the node two steps ahead. In the next round, every node asks the same question of that node two steps ahead, and learns about the node four steps ahead. Then eight, sixteen, and so on: the jump doubles at every round. Attention is perfectly suited to this, since each token can look up any other token in a single layer. In principle, with L layers, a model can therefore follow chains of up to about $2^L - 1$ hops: 7 with 3 layers, 15 with 4, 31 with 5, 63 with 6.

Ok now, let's have a look at the results! The figure below represents the accuracy as a function of the number of hops necessary, for looped models on the left (with more and more looping) and vanilla models on the right (with more and more layers, hence parameters).

Accuracy versus number of hops, for looped and deep models applying 3 to 6 layers

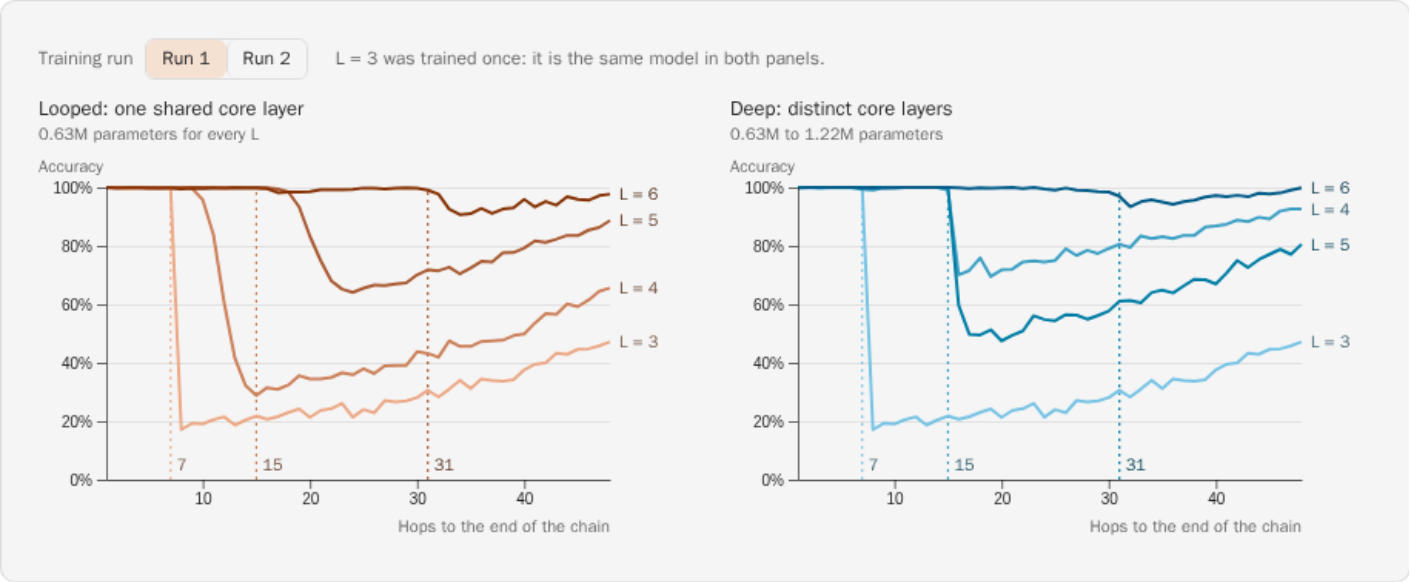


Figure 3 · Each line is one model, and L is the number of layers it applies per token. The dotted vertical lines mark $2^L - 1$ hops, the reach of pointer doubling with L layers. Every chain length up to 48 hops was seen in training. The buttons switch to the second training run.

We can see the 3-layer model follows chains of exactly 7 hops, then collapses, just as predicted. With 4 layers, the deep model reaches 14 to 15 hops (over two training runs) and the looped one 10 to 11. With 5 layers, both fall short of the theoretical 31: 16 to 19 hops for the looped model, 15 for the deep one. With 6 layers, both solve every chain up to 48 hops.

Something that might seem surprising : for longer chains the performance seems to recover. How is it possible ?? Actually it comes from what we could call an *elimination artefact*: on longer chains, few other chains are left, so the model manages to guess the answer more easily. We can't really consider it as an actual improvement of the raw performance.

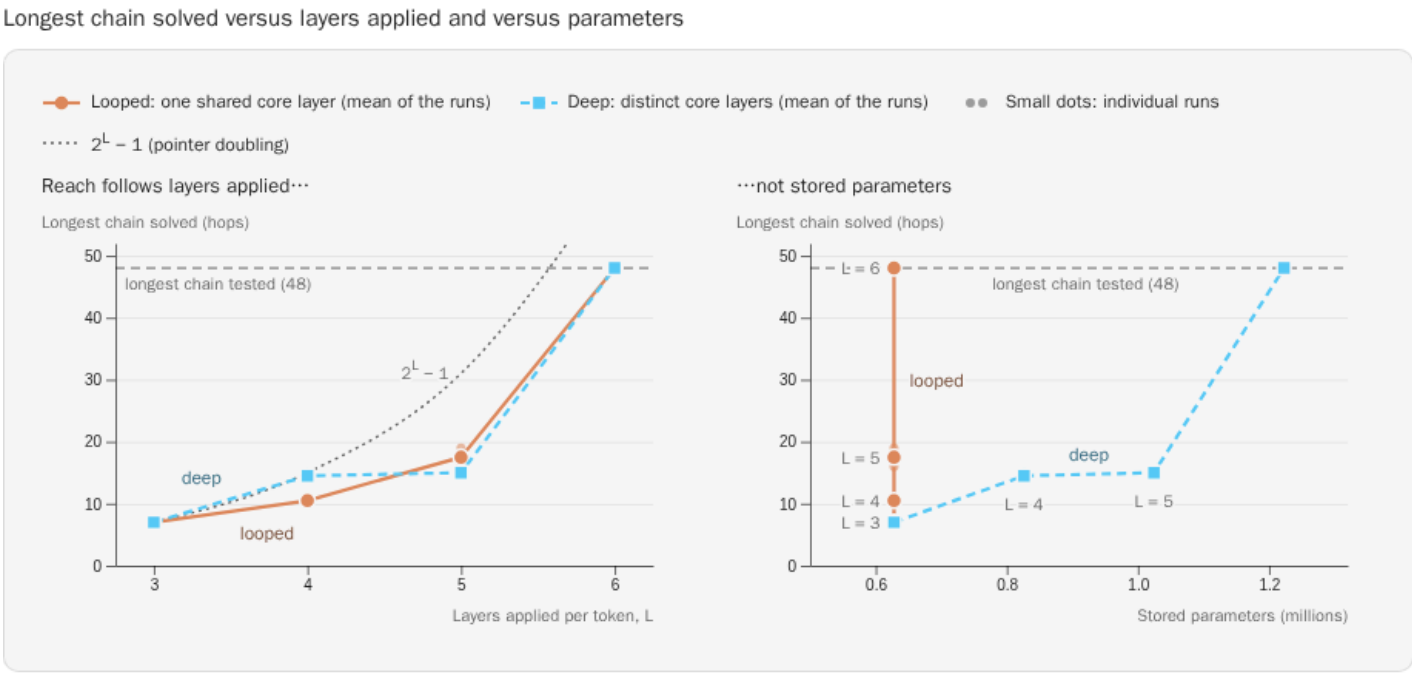


Figure 4 · The longest chain solved is the largest number of hops such that every chain up to that length is answered with at least 90% accuracy. Large markers show the mean of the two training runs, small dots each run (the model with $L = 3$ was trained once).

This summary figure tells the story. On the left, the longest chain solved against the number of layers applied: the two families track each other, the looped one ahead at 5 layers and behind at 4. On the right, the same results against the number of stored parameters: the looped models go from 7 to 48 hops without gaining a single parameter. For following a chain, what matters is how many layers are applied, not how many are stored. That is exactly the bet behind looped models.

2.3 Memorising facts: where parameters win

Now the opposite question. Most of what a large language model knows is not in its input: the capital of Peru, the year a paper came out. It has to be stored in the weights. Does looping help there? Probably not!

To test this, I took three models with a two-layer core:

- shallow: the core runs once (4 layers applied, 0.83M parameters);
- looped: the core runs 4 times (10 layers applied, 0.83M parameters);
- deep: 8 distinct core layers (10 layers applied, 2.0M parameters).

I trained each of them to memorise a table of one million random facts. In my simple experiment, a fact links a “subject” (four symbols) to an “answer” (one of 64 symbols). The facts are random, so there is nothing to reason about and nothing to generalise: a fact is

really either stored in the weights or it isn't. A million facts is more than any of these models can hold, so the question is how many each one keeps.

Here are a few rows of the table, with the four symbols of the subject on the left and the answer on the right:

```
subject          answer

47 44 54  1  →   34

47 44 54 21  →   19

19 13 46 25  →   37

19 13 46 48  →   33

19 13 46 16  →   38

38 25 40 16  →   50

...              (1,000,000 rows in total)

```

Think of each row as a fact like "the capital of Peru is Lima", written in a code where the words mean nothing and the answers were drawn at random. Subjects that share three symbols out of four have unrelated answers, so there is no pattern to exploit: the only way to get a row right is to have memorised it.

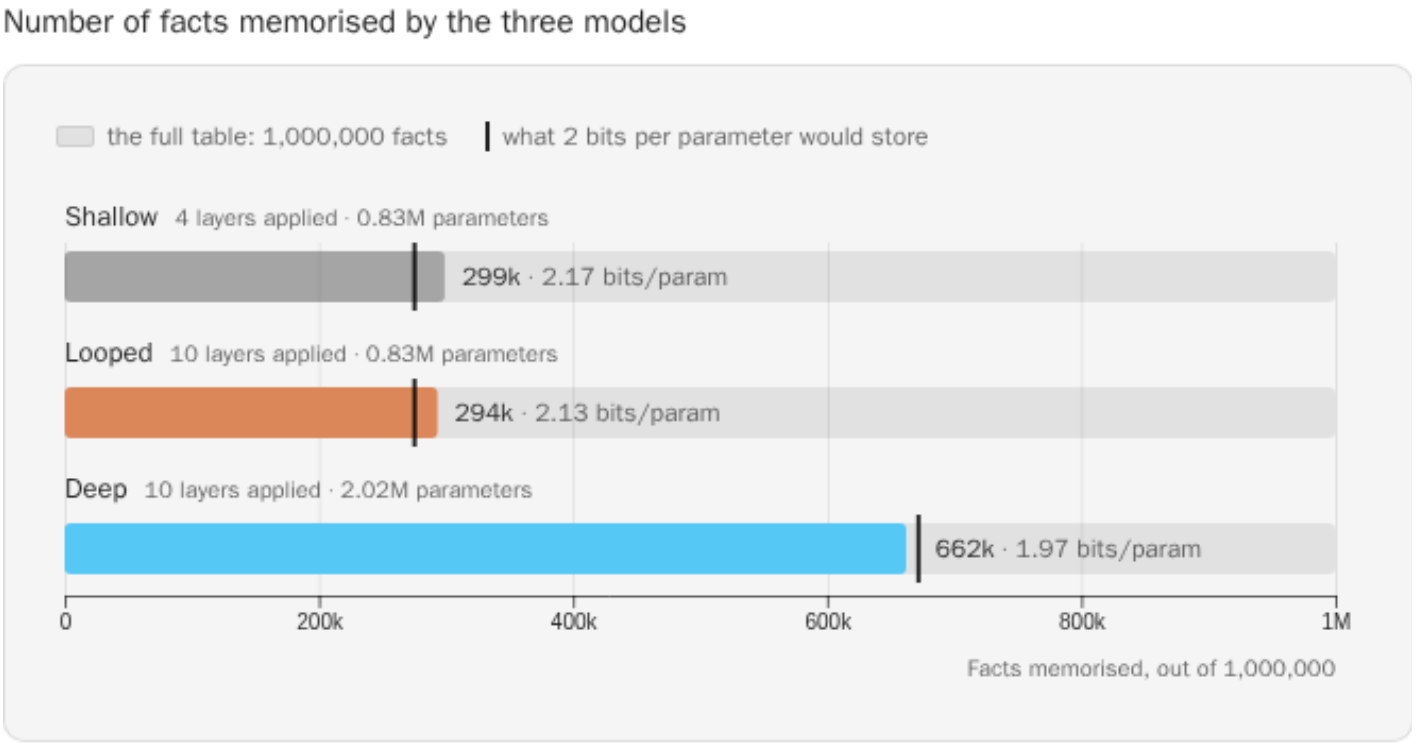


Figure 5 · All three models were trained on the same table of one million random facts. The vertical tick on each bar shows how many facts 2 bits per parameter would hold, at 6 bits per fact.

The shallow and looped models store almost the same number of facts: 299,000 and 294,000. The looped model applies two and a half times as many layers, and it makes no difference at all. The deep model, with 2.4 times as many parameters, stores 662,000.

Divide the stored information by the number of parameters (each answer is worth 6 bits) and you get about 2 bits per parameter for all three models: 2.17, 2.13 and 1.97. That is close to what Allen-Zhu and Li measured in language models in 2024 (Allen-Zhu & Li, 2024), about 2 bits of knowledge per parameter, whatever the architecture. Looping doesn't change that number.

Knowledge lives in the parameters, and passes don't add parameters. Put this next to the previous result and the picture is clear: looping gives you more reasoning steps for the same memory, but no extra memory.

2.4 Choosing the number of passes at test time

So far, each looped model was trained and tested with a fixed number of passes. Inkling's research variant does something more interesting: it was trained with a random number of passes (1 to 5), so you can choose how many passes to run when you use it. Let's do the same!

I trained a model with a two-layer core, drawing the number of passes at random between 1 and 6 at each training step. All training chains had at most 16 hops. Then I tested it on chains of up to 48 hops, with anywhere from 1 to 16 passes.

Accuracy versus number of hops for the variable-loop model, for different numbers of passes at test time

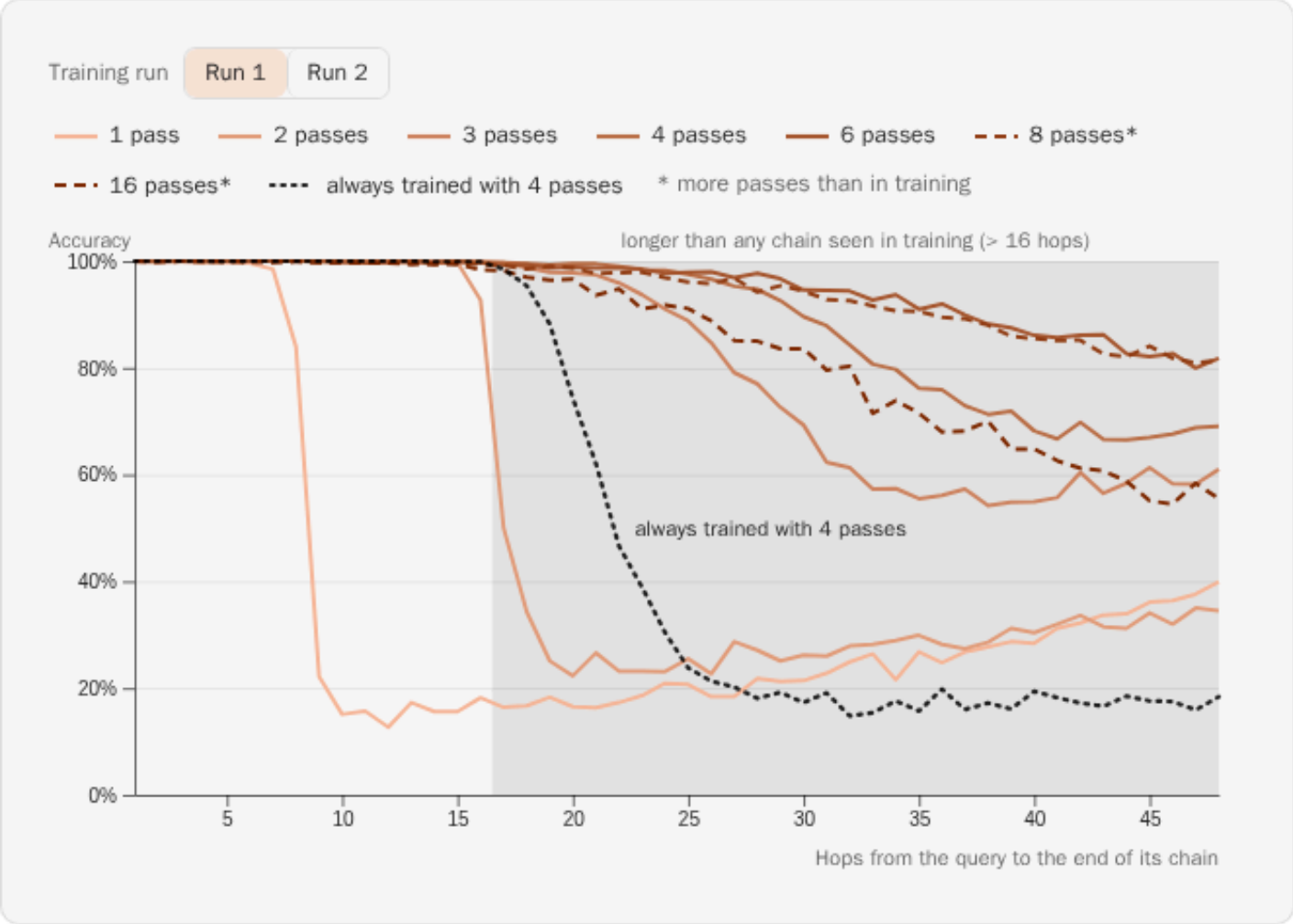


Figure 6 · One model, trained with 1 to 6 passes on chains of at most 16 hops. Dashed lines use more passes than in training. The dotted line is the same architecture always trained with 4 passes (one training run). Hover over the legend to isolate a curve.

Two interesting things can be seen here. First, each extra pass extends the reach. With one pass, the model follows chains of up to 7 hops (at 90% accuracy or more). With two passes, 16. With three, 24. With six, 36. And with seven passes (one more than it ever saw in training), 38 hops. Remember that no training chain was longer than 16 hops: from three passes on, the model is solving problems longer than anything it was trained on.

Second, this is not just a matter of compute. The dotted line is a model with exactly the same architecture, always trained with 4 passes. Run at 4 passes, it uses exactly as much compute as the variable-loop model run at 4 passes. Yet it breaks down at 18 hops, while the variable-

loop model reaches 29. It seems that training with a random number of passes forces the model to learn a procedure that keeps working when you repeat it, and such a procedure also carries over to longer chains (see the next section).

There is a limit, though. Beyond six or seven passes, the reach stops growing. In the first run it even *decreases* (31 hops at 10 passes, 25 at 16); in the second it stays flat around 30 hops all the way to 16 passes. Either way, running the loop much longer than in training buys nothing, and it can even hurt (we'll see one reason in a moment.)

2.5 Watching the loop work

Because this model was trained to answer after any number of passes, we can do something you can't do with an ordinary network: stop after pass 1, 2, 3⋯ and read the model's answer at each stage. (Thinking Machines' variable-loop model allows exactly the same thing.)

The figure below shows, for chains of 16, 24 and 32 hops, how much probability the model gives to each node along the query's chain after each pass. Position 0 is the query, the last position is the correct answer.

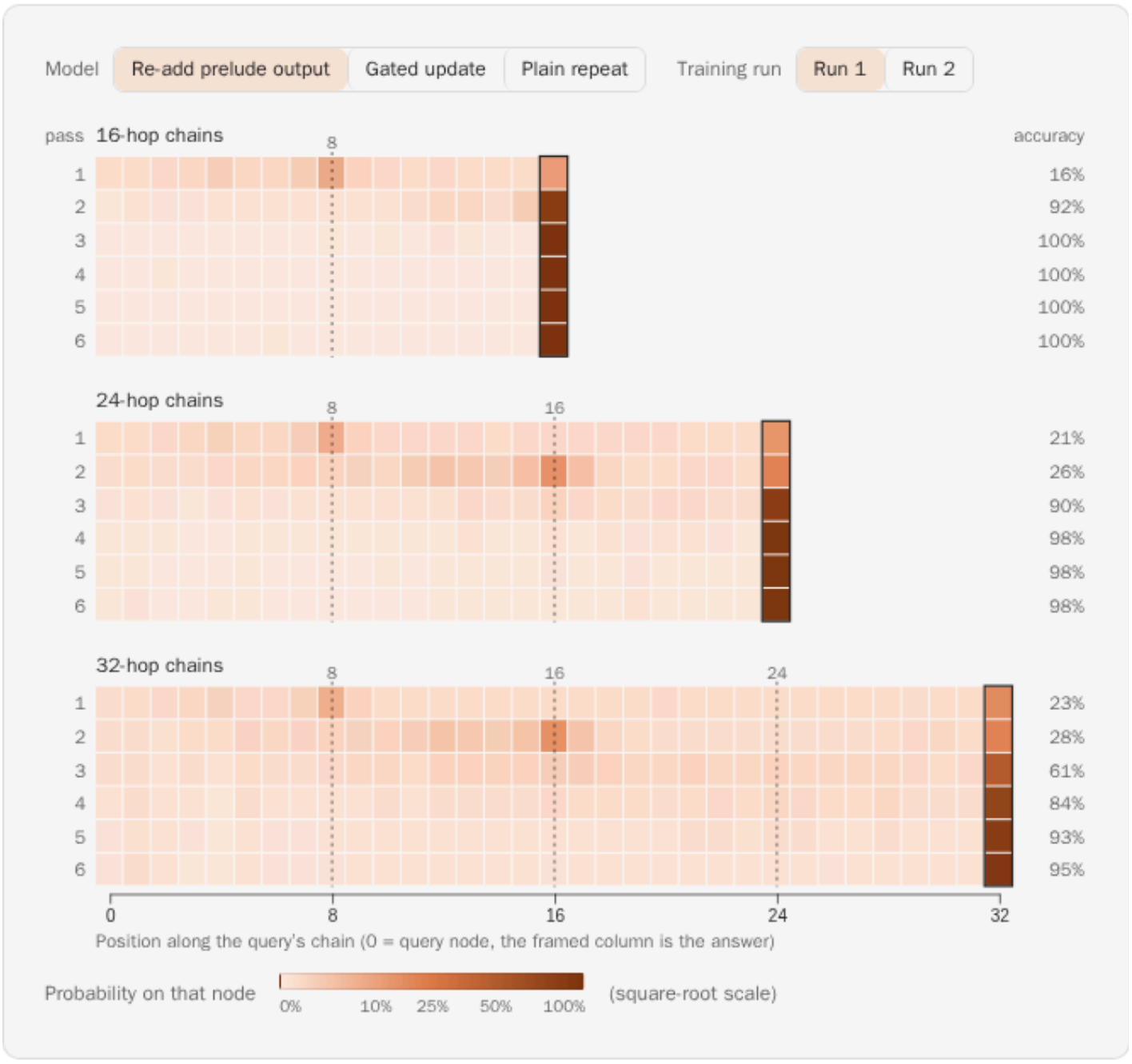


Figure 7 · Rows are passes and columns are positions along the query's chain (0 is the query, the framed column is the answer). Darker cells get more probability. The column on the right gives the accuracy after each pass. The text discusses the first run of the model that re-adds the prelude output; the buttons show its second run and the two other variants of the last section.

Look at where the probability concentrates when the model hasn't finished. After pass 1, it points at the node 8 hops down the chain. After pass 2, at the node 16 hops down. On 24-hop chains, it reaches the end at pass 3. On 32-hop chains (twice as long as anything seen in training), it gets there partially at pass 3 and mostly at pass 4. In this run, each of the first passes doubles how far the model can see.

This is pointer jumping again, the trick from the depth experiment, but now we can watch it happen. The model was never told about this algorithm. It found it, and it uses each pass through its core to perform one doubling.

I should be honest about the second training run: there the picture is blurrier. The intermediate guesses also move forward pass after pass (about 13, then 15, then 21 hops down the chain), but without the clean 8-then-16 pattern. So doubling is one solution that training can find, not the only one.

The end of a chain is also a natural resting point: once the model points at it, there is nowhere further to go, and the answer stays put. That is why extra passes do little harm, at least at first.

2.6 Keeping the loop stable

Now let's consider the engineering problem Z.ai describes for GLM-5.4: what happens to the internal state when you loop many times?

I compared three ways of connecting one pass to the next:

- Plain repeat: the output of pass *t* is the input of pass *t+1*, and that's it.
- Re-add the prelude output: at every pass, the prelude's output is added back to the state before it enters the core. This is the variant used in all the previous figures (Inkling also restarts every pass from the prelude output).
- Gated update: the core proposes a new state, and a small learned gate, one number per token, decides how much of it to write (0 = keep the old state, 1 = replace it). This is my simplified version of GLM-5.4's gates. One difference is that GLM-5.4 has separate gate parameters for each pass, while mine are shared across passes so that the model can run any number of passes.

All three were trained the same way, with 1 to 6 random passes.

Reach, state norm and gate values for the three variants

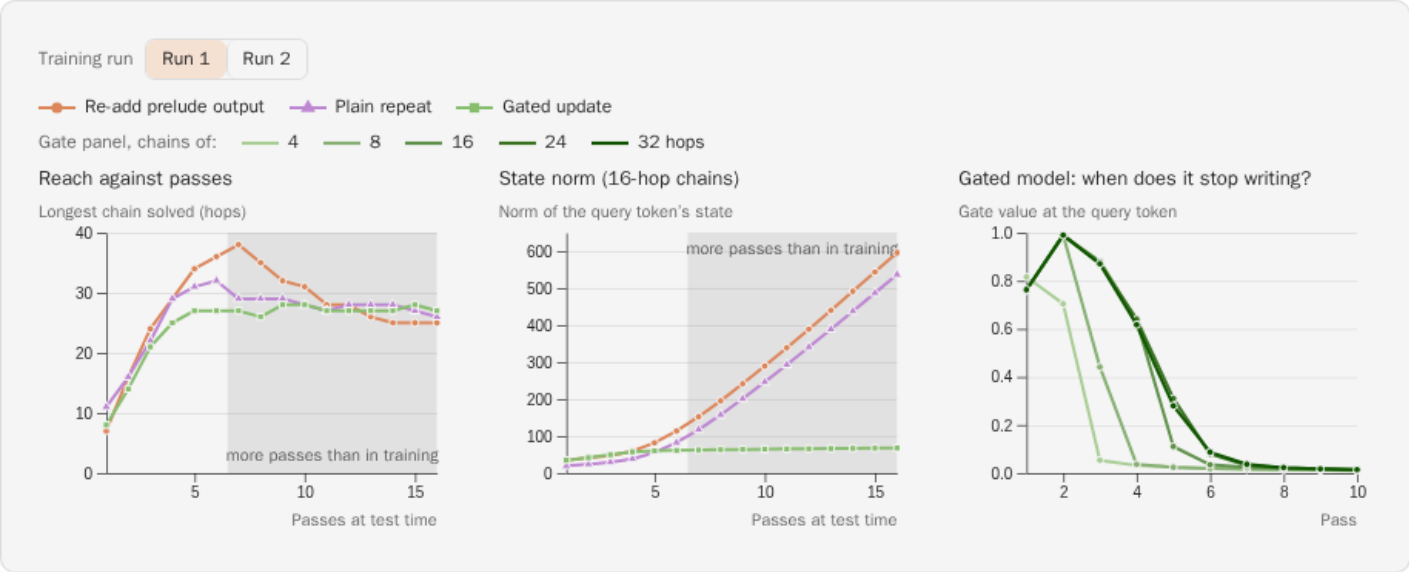


Figure 8 · Left: longest chain solved (at 90% accuracy) against the number of passes at test time. Middle: norm of the query token's internal vector on 16-hop chains. Right: gate value at the query token in the gated model, for chains of 4 to 32 hops. Shaded areas: more passes than in training (1 to 6).

The middle panel shows the size (norm) of the query token's internal vector after each pass. Without a gate, it grows steadily, by about 50 per pass once the answer has been found, reaching 17 to 27 times its initial size after 16 passes. Nothing stops it because every pass adds its contribution on top of the previous ones. This is the blow-up Z.ai describes, *"activation norms blow up across iterations"*, reproduced in a model with less than a million parameters. It is a plausible cause of the decline of the reach beyond 7 passes in the first run: the later layers receive vectors much larger than anything they saw during training. (It is not always fatal, though in the second run the norm grows just as much from 39 to 710 and the reach holds.)

With the gate, the norm stays flat in both runs, and the reach never degrades: 26 to 28 hops from 5 passes all the way to 16 (24 to 25 in the second run). The price is that the gated model reaches less far at its best (28 hops versus 38).

Now look at the right panel. It shows the gate value at the query token, pass after pass, for chains of different lengths. The gate is wide open during the first passes, then drops to almost zero, and it closes later for longer chains: it falls below 0.1 at pass 3 for 4-hop chains, at pass 4 for 8-hop chains, and at pass 6 for chains of 16 hops or more. Nobody asked the model to do this! It learned on its own to keep working while the answer is not yet found, and to stop writing once it is. That is a form of *adaptive computation* (spending more steps on harder inputs), which is precisely the open question Z.ai lists at the end of its GLM-5.4 announcement.

The second run shows the same tendency, but cruder: the gate closes at pass 2 for 4-hop chains, and at pass 3 for every chain of 8 hops or more. It stops adapting beyond that point, and that model never gets past 25 hops. A gate that closes too early caps how far the model can go.

A word of caution from initial attempts. I first tried an Inkling-style gate that starts *fully closed* (the message from the previous pass is multiplied by zero at the beginning of training). In 2,000 training steps it opened to about 0.06, and the looped model ended up no better than a non-looped one with the same parameters: it essentially ignored its extra passes. Inkling's gate had a full-scale training run to open, but it seems that on a small budget like mine, a closed gate can stay closed.

2.7 Conclusion: What the tiny models say about the big ones

Let's put the small-scale results next to what the two labs report, to see whether we learned something useful with these tiny experiments.

Extra passes help multi-step reasoning. In my toy experiment: the chain length a model can follow depends on how many layers it applies, not on how many it stores. On Inkling's variable-loop model, going from 1 to 3 passes takes GraphWalks (walking through a graph of up to a million tokens given in the context) from 5.7 to 52.1, BrowseComp from 29.4 to 65.7, and Humanity's Last Exam from 10.5 to 24.9.

Extra passes do not add knowledge. In my toy experiment: the number of facts a model memorises depends on its parameters, not on its passes. On the same Inkling model, SimpleQA Verified, a benchmark of short factual questions whose answers must be stored in the weights, goes from 6.3 to 7.3 between 1 and 3 passes. Almost nothing. Z.ai, for its part, reports its largest gains in math and scientific reasoning.

The returns stop at some point. In my toy experiment: reach stops improving a few passes beyond the number used in training. Inkling's variable-loop model was trained with up to 5 passes but post-trained with 3, and its scores barely move beyond 3 (Humanity's Last Exam: 24.9, 25.4, 25.9 at 3, 4 and 5 passes).

Looping needs a mechanism to stay stable. In my toy experiment: without a gate, the internal state grows with every pass. Z.ai built GLM-5.4's per-token gates for exactly this reason, and Inkling's gate controls how much of the previous pass gets through.

At equal compute, there is no free lunch. In my toy experiment: a non-looped model applying the same number of layers follows chains just as well *and* stores more facts. That is consistent with Z.ai's statement that looped models trained from scratch lag behind at equal training compute. Thinking Machines frames the trade-off in the same way: their non-looped comparison model was cheaper to train, and the looped model's advantage showed up at inference time. Looping pays off when the scarce resource is *memory*, the weights that must fit on a laptop or a GPU, rather than computation.

One thing I can't reproduce with my tiny experiment: both labs report that *looped models write shorter reasoning* (on GPQA, Inkling's variable-loop model goes from 3,703 reasoning tokens at 1 pass to 1,731 at 3 passes; GLM-5.4's outputs are 10 to 14% shorter). My toy models answer in a single token, with no written reasoning at all (a natural extension would be to let them write intermediate nodes, and count how many they need at each number of passes)

(Benchmark numbers are from the TML-d24-varloop-research model card; check them against the card before publishing.)

That being said, keep in mind that my models have about a million parameters and see a single synthetic task. Also their attention is bidirectional over an unordered set, while language models read text from left to right. So take these results as illustrations of mechanisms, not as measurements of what happens in a 26B model!

A few questions remain open at this stage (I guess even for the labs):

- How many passes does each token need? Both releases use a fixed number of passes for every token. Z.ai explicitly lists adaptive computation as future work. The gate in my small models suggests that a model can learn to stop on its own… but whether that survives at scale is another matter!
- Can we still read what the model thinks? If part of the reasoning moves from written tokens into the loop, it becomes harder to monitor. Thinking Machines studied this and found no meaningful loss of monitorability at this scale, while warning that this may not last. This might be a concern for interpretability.
- Why do looped models trained from scratch lag behind? In my toy experiment, at equal compute, looping loses nothing on reasoning and only loses storage. Whether that is the whole explanation for large models, I don't know.
- How do returns evolve with model size and number of passes? Z.ai lists this as open too.

2.8 Reproducing the experiments

Everything is in a single script, `loopexp.py`, which runs on Hugging Face Jobs with no setup:

```
hf jobs uv run --flavor a10g-small loopexp.py law --train_hops 48
# depth sweep
hf jobs uv run --flavor a10g-small loopexp.py facts --K 1000000 --fact_steps
50000         # memorisation
hf jobs uv run --flavor a10g-small loopexp.py varloop --only inject
# variable-loop model
hf jobs uv run --flavor a10g-small loopexp.py varloop --only gated,plain
# the two other variants
```

The runs behind the figures in this post took about 3.5 GPU-hours on A10G GPUs, roughly 3.50 USD on Hugging Face Jobs at the time of writing.

If you want to read the code before running it (or adapt it to your own task), here it is in full. Click to unfold it, and use the copy button in its top-right corner.

2.9 Sources

- Thinking Machines, Introducing Inkling 2 Mini, Inkling 2 Mini model card, TML-d24-varloop-research
- Hugging Face, Welcome Inkling 2 Mini
- Z.ai, GLM-5.4 overview
- VentureBeat, Z.ai's GLM-5.4 ⋯ running 28 layers twice
- Goodfire, Interpreting Inkling 2 Mini

Citation

For attribution in academic contexts, please cite this work as

```
David Louapre (2026). "Looped Transformers: Thinking twice with the same weights".
```

BibTeX citation

```
@misc{louapre2026_looped_transformers_thinking_twice_with_the_same_weights,
  title={Looped Transformers: Thinking twice with the same weights},
  author={David Louapre},
  year={2026},

}
```

Reuse

Diagrams and text are licensed under CC-BY 4.0, unless noted otherwise. Figures reused from other sources are excluded and marked in their captions (“Figure from ⋯”).

References

1. Allen-Zhu, Z., & Li, Y. (2024). *Physics of Language Models: Part 3.3, Knowledge Capacity Scaling Laws*. https://arxiv.org/abs/2404.05405↑
2. Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., & Kaiser, Ł. (2018). *Universal Transformers*. https://arxiv.org/abs/1807.03819↑
3. Geiping, J., McLeish, S., Jain, N., Kirchenbauer, J., Singh, S., Bartoldson, B. R., Kailkhura, B., Bhatele, A., & Goldstein, T. (2025). *Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach*. https://arxiv.org/abs/2502.05171↑
4. Giannou, A., Rajput, S., Sohn, J., Lee, K., Lee, J. D., & Papailiopoulos, D. (2023). *Looped Transformers as Programmable Computers*. https://arxiv.org/abs/2301.13196↑
5. Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019). *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. https://arxiv.org/abs/1909.11942↑
6. Saunshi, N., Dikkala, N., Li, Z., Kumar, S., & Reddi, S. J. (2025). *Reasoning with Latent Thoughts: On the Power of Looped Transformers*. https://arxiv.org/abs/2502.17416↑
7. Zhu, R.-J., Wang, Z., Hua, K., Zhang, T., Li, Z., Que, H., Wei, B., Wen, Z., Yin, F., Xing, H., Li, L., Shi, J., Ma, K., Li, S., Kergan, T., Smith, A., Qu, X., Hui, M., Wu, B., ⋯ Eshraghian, J. (2025). *Scaling Latent Reasoning via Looped Language Models*. https://arxiv.org/abs/2510.25741↑

Made with ❤️ with research article template