Modular Connections: The Connections Between Residual Streams Can Themselves Be Modular Models

Yue Su

November 6, 2026 · DOI: 10.5281/zenodo.24032654
Web version with interactive animations: https://selen-suyue.github.io/modular-connections/en/

Abstract

Manifold-Constrained Hyper-Connections (mHC) widen the residual stream of a Transformer into N parallel streams and, at every sublayer, generate per-token coefficients that decide which streams to read from, which streams to write to, and how the streams mix. I show that once the N streams are interleaved, these three operations are exactly a convolution and a deconvolution with kernel size and stride N, and a sliding N×N map, so the connections of each token are simply a set of kernels. Building on this view, I propose Modular Connections (MoC): every sublayer keeps banks of read, write and mixing experts, and each token is routed to a few of them, as in a mixture of experts, and combines them with weights. This design matches a recent analysis of DeepSeek-V4-Flash, whose trained mHC settles on fairly fixed connection patterns per sublayer with little per-token variation. Because the mixing matrices of MoC are expert parameters rather than per-token outputs, their Sinkhorn normalization can be precomputed after training. In my measurements on a 1.36B-parameter model, the MoC forward pass is 2.2–3.2× faster than that of mHC on single sequences without compilation and about 1.5× faster on short inputs once compiled, the training step compiles about 16× faster, and routing can be inspected expert by expert. In an ongoing 49B-token run, the validation bits-per-byte of MoC has stayed within −0.0009 to +0.0017 of mHC at matching steps from step 2,000 to step 6,360.

This PDF is a static print of the web version. Each animation is shown as one frame with its step caption; the web version lets you play and step through them.

DeepSeek's mHC (Manifold-Constrained Hyper-Connections) widens the Transformer's single residual stream into N parallel streams. Every sublayer then has to answer three questions: which streams to read its input from, which streams to write its output back to, and how the streams mix with one another. Taken together, these are the connections between the residual streams.

In mHC, these connections are only a few sets of coefficients: each token passes through a small projection (plus static biases) that computes its read, write and mixing weights on the spot. My finding is that **the connections between residual streams can themselves be modular models**. Each sublayer can maintain a bank of read, write and mixing modules, and each token is routed to a few of them, as in a mixture of experts, and combines them with weights. Where MoE modularizes the computation inside a sublayer, this modularizes the connections between residual streams. I call the approach **Modular Connections**, or **MoC**.

A recent analysis of DeepSeek-V4-Flash supports this view from the opposite direction (Zhao et al., arXiv:2609.05309). Although that model's mHC could produce different connections for every token, the trained model uses little of this freedom. The effective number of streams a sublayer reads from or writes to averages about two, and within an input sequence nearly 90% of tokens share the same dominant stream. Fixing the mixing matrices in the first 22 of its 43 layers to their averages over the data raises perplexity by only 0.2%, and the mixing matrices in the last 21 layers are close to the identity. In other words, **the connections are already modular**: each sublayer settles on fairly stable connection patterns, and mHC's per-token generation of coefficients obscures this.

The idea came to me while working through mHC's computation for a single token on paper. If the N streams are laid out in a different order, read, write and mix become exactly a convolution, a deconvolution (transposed convolution) and a sliding 4×4 map (Section 2). From this angle, each token's connections are simply a set of kernels, and nothing

prevents a model from keeping many kernels and letting each token choose among them. Section 3 describes MoC, Section 4 uses it to reinterpret the observations on DeepSeek-V4-Flash, and Section 5 reports what MoC gains in practice.

---

1. One step of mHC

A standard residual connection carries a single stream:

$$x_{l+1} = x_l + F(x_l)$$

Hyper-Connections widen the C-dimensional residual state into N streams, written $X_l \in \mathbb{R}^{N\times C}$; mHC uses $N = 4$. Each sublayer (Attention or MLP) performs three operations on these streams:

- **Read** with $H_{\text{pre}} \in \mathbb{R}^{1\times N}$: combine the N streams into one with weights, which becomes the input of sublayer F;
- **Write** with $H_{\text{post}} \in \mathbb{R}^{1\times N}$: add the output of F back to every stream with weights;
- **Mix** with $H_{\text{res}} \in \mathbb{R}^{N\times N}$: let the streams mix with each other.

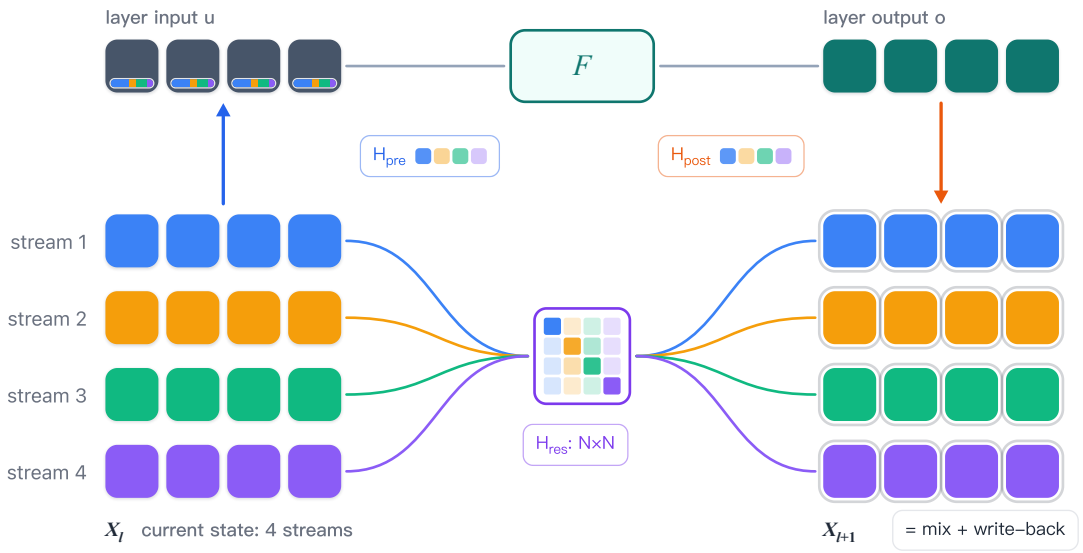
$$X_{l+1} = H_{\text{res}}\, X_l + H_{\text{post}}^{\top}\, F(H_{\text{pre}}\, X_l)$$

In mHC, all three sets of coefficients are generated from the current token's state: $H_{\text{pre}} = \sigma(\cdot)$, $H_{\text{post}} = 2\sigma(\cdot)$, and H_{res} is projected onto the doubly stochastic matrices by Sinkhorn–Knopp. For any single token, then, these coefficients are just a fixed set of numbers.

The animation below walks through the computation for one token. For readability it uses $N = 4$ streams of $C = 4$ dimensions each.

stream 1 stream 2 stream 3 stream 4 layer input (bottom bar = share from each stream) layer output

Animation 1 One mHC step for a token (stream view)



Step 4 / 4 · Mix and add

Mix: H_{res} is an N×N matrix that mixes the 4 streams with each other. The mixed result plus the written-back part gives the next layer's state X_{l+1}.

mHC step $$X_{l+1} = H_{\text{res}}\, X_l + H_{\text{post}}^{\top}\, F\,(\, H_{\text{pre}}\, X_l\,)$$

2. The finding: one mHC step is a set of convolutions

The key: an interleaved layout

Take $N = 4$ streams of $C = 4$ dimensions each, and interleave them into a 16-dimensional vector v so that position $c{\cdot}N + i$ holds dimension c of stream i:

$$v[c{\cdot}N + i] = X_l[i, c]$$

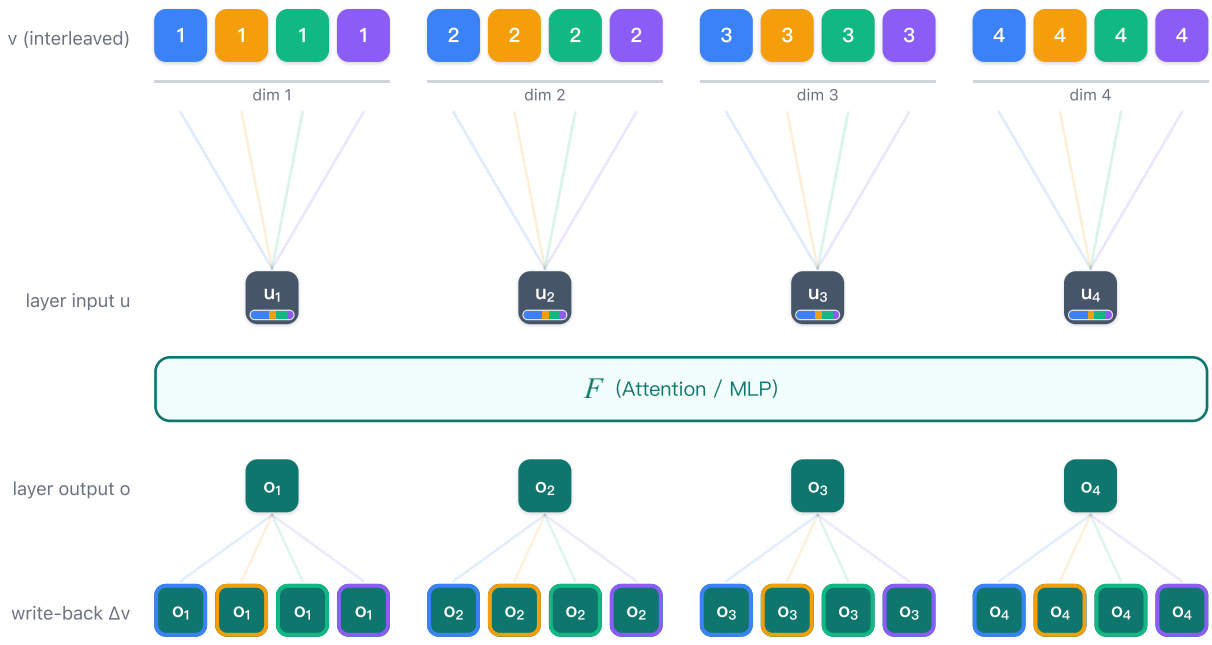
stream 1 → positions 1, 5, 9, 13; stream 2 → positions 2, 6, 10, 14; streams 3 and 4 likewise

With this layout, every 4 adjacent positions hold the same dimension of all 4 streams. Since each of mHC's three operations only ever combines the 4 streams of a single dimension, all three become operations on this vector with a window and stride of 4:

- **Read = convolution.** The kernel is the 4 weights of h_{pre}, with kernel = stride = 4. Each window outputs one number $u_c = \Sigma_i\, h_{\text{pre},i} \cdot x_{i,c}$; sliding it 4 times yields the 4-dimensional sublayer input.
- **Write = deconvolution.** The kernel is the 4 weights of h_{post}, with kernel = stride = 4. Each output o_c of F is expanded into the 4 values $h_{\text{post},i} \cdot o_c$ of window c.
- **Mix = a sliding 4×4 map.** The same 4×4 linear map H_{res} is applied to each window in turn, moving 4 times. It is equivalent to a convolution with 4 output channels and kernel = stride = 4.

Adding the two branches gives the next layer's state. The animation below goes through this step by step; you can pause it, step through it, or click the dots underneath to jump to any step.

Animation 2 The same step of the same token, seen as convolutions



Step 4 / 6 · Write = deconvolution

Write is a deconvolution: the kernel is the 4 weights of h_{post}, with kernel = stride = 4. Each output o_c is expanded into the 4 positions of window c: $\Delta v = h_{\text{post},i} \cdot o_c$, i.e. written back to dimension c of the 4 streams.

Layout $v[c \cdot N + i] = X_l[i, c]$

Streams $X_{l+1} = H_{\text{res}} X_l + H_{\text{post}}^{\top} F(H_{\text{pre}} X_l)$

Convolutions $v_{l+1} = \text{Mix}_{H_{\text{res}}}(v) + \text{Deconv}_{h_{\text{post}}}(F(\text{Conv}_{h_{\text{pre}}}(v)))$

In short, for each token and given its coefficients,

$$X_{l+1} = H_{\text{res}} X_l + H_{\text{post}}^{\top} F(H_{\text{pre}} X_l)$$

$$\Longleftrightarrow$$

$$v_{l+1} = \text{Mix}_{H_{\text{res}}}(v) + \text{Deconv}_{h_{\text{post}}}(F(\text{Conv}_{h_{\text{pre}}}(v)))$$

The equivalence holds under three conditions: v is interleaved as above; the convolution and the deconvolution both have kernel size and stride equal to N, so the windows do not overlap; and the sliding 4×4 map is purely linear.

Two algorithms, one result

Below, the same step is computed both ways on random inputs. The middle column uses the original mHC formula; the right column uses the convolution view, interleaving, convolving, applying F and deconvolving, adding the sliding 4×4, and finally splitting the result back into 4 streams. Here F is a random "linear layer + tanh".

Animation 3 Numeric check (each click draws new random numbers)

Input X_l: 4 streams × 4 dims

−0.632	−0.303	0.796	−0.996
−0.326	0.767	0.220	0.123
−0.194	0.487	0.094	0.471
0.669	0.904	0.928	−0.896

Stream view: $H_{\text{res}}X + H_{\text{post}}^{\top}F(H_{\text{pre}}X)$

−0.044	0.260	0.794	−0.550
−0.285	−0.011	0.666	−0.155
−0.349	−0.574	1.005	−0.281
−0.128	−0.874	1.447	−0.634

Convolution view, split back into 4 streams

−0.044	0.260	0.794	−0.550
−0.285	−0.011	0.666	−0.155
−0.349	−0.574	1.005	−0.281
−0.128	−0.874	1.447	−0.634

Result in the convolution view, v_{l+1} (16 dims, interleaved)

−0.044	−0.285	−0.349	−0.128	0.260	−0.011	−0.574	−0.874
0.794	0.666	1.005	1.447	−0.550	−0.155	−0.281	−0.634

Largest difference between the two algorithms: **0 (bit-for-bit identical)** ✓

The same check takes a few lines of PyTorch, where the kernels of `conv1d` and `conv_transpose1d` are exactly mHC's coefficients:

```
import torch, torch.nn.functional as F

N, C = 4, 4
x = torch.randn(N, C)                       # x[i, c]: dimension c of stream i
h_pre, h_post = torch.rand(N), torch.rand(N)
H_res = torch.rand(N, N)
f = torch.tanh                              # any layer function

# stream view: the mHC formula
ref = H_res @ x + h_post[:, None] * f(h_pre @ x)[None, :]

# convolution view: interleave, position c*N + i holds x[i, c]
v = x.T.reshape(1, 1, N * C)
u = F.conv1d(v, h_pre.view(1, 1, N), stride=N)                  # read: convolution
dv = F.conv_transpose1d(f(u), h_post.view(1, 1, N), stride=N)   # write: deconvolution
m = F.conv1d(v, H_res.view(N, 1, N), stride=N)                  # mix: sliding 4×4
m = m.transpose(1, 2).reshape(1, 1, N * C)
out = (m + dv).view(C, N).T                                     # split back into 4 streams

print(torch.allclose(out, ref, atol=1e-6))                      # True
```

---

3. The proposal: Modular Connections (MoC)

In the convolution view, every token in mHC "generates" its own kernels: a read convolution kernel h_{pre}, a write deconvolution kernel h_{post}, and a sliding 4×4 map H_{res}.

MoC **turns these connections into modules**. Rather than having each token generate its kernels directly, the model learns a set of convolution / deconvolution experts and 4×4 mixing experts; each token is routed to a few of them, as in a mixture of experts, and combines them with weights.

Three groups of experts

As in MoE, every sublayer has its own expert bank, organized into three groups:

- **Read convolution experts** $\{K_{\text{pre}}^{(1)}, \ldots, K_{\text{pre}}^{(E)}\}$: each is a convolution kernel of length N;
- **Write deconvolution experts** $\{K_{\text{post}}^{(1)}, \ldots, K_{\text{post}}^{(E)}\}$: each is a deconvolution kernel of length N;
- **4×4 mixing experts** $\{M^{(1)}, \ldots, M^{(E)}\}$: each is an $N \times N$ map that slides along the 16-dim vector.

All experts are learned parameters shared by every token. The mixing experts are still normalized with Sinkhorn–Knopp, but the normalization now acts on the expert parameters themselves: during training it is recomputed after each parameter update and shared by all tokens, and after training it can be computed once and stored.

Building the mixing matrix as a combination of other matrices is not new: mHC-lite writes it as a convex combination of fixed permutation matrices and thereby avoids Sinkhorn altogether (Yang and Gao, arXiv:2601.05732). MoC differs in that read, write and mix are all learnable experts, and each group is routed sparsely on its own.

Routing and weighting

For each token, a router scores the experts in every group from the token's state, keeps the k highest-scoring ones, and applies a softmax over their scores to obtain the weights g:

$$s = \mathrm{Router}(x_t), \quad R = \mathrm{TopK}(s, k), \quad g = \mathrm{softmax}(s_R)$$

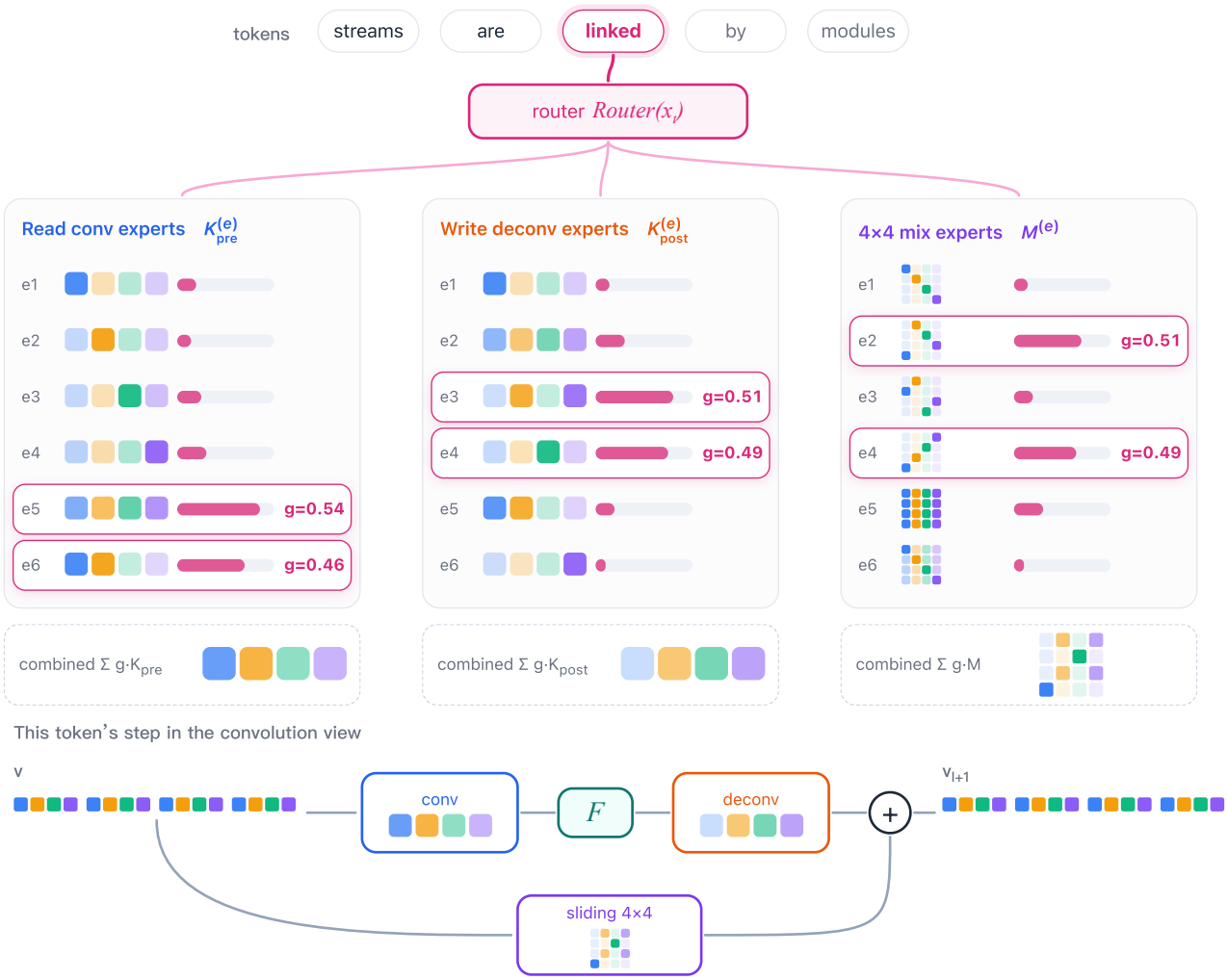
The selected experts then carry out the step:

$$\begin{aligned} u &= \Sigma_{e \in R_{\mathrm{pre}}} g_e \cdot \mathrm{Conv}(v;\, K^{(e)}_{\mathrm{pre}}) \\ \Delta v &= \Sigma_{e \in R_{\mathrm{post}}} g_e \cdot \mathrm{Deconv}(F(u);\, K^{(e)}_{\mathrm{post}}) \\ m &= \Sigma_{e \in R_{\mathrm{res}}} g_e \cdot \mathrm{Mix}(v;\, M^{(e)}) \\ v_{l+1} &= m + \Delta v \end{aligned}$$

Because convolution is linear in its kernel, $\Sigma_e\, g_e \cdot Conv(v;\, K^{(e)}) = Conv(v;\, \Sigma_e\, g_e\, K^{(e)})$, the selected kernels can first be merged into a single weighted kernel and applied with one convolution; the same holds for the deconvolution and the mixing. Since the weights g are non-negative and sum to 1, the merged mixing matrix is a convex combination of the selected experts.

The animation below processes three tokens in turn, following the order “score, pick the top k, combine, plug in”. For illustration, each group has $E = 6$ experts and $k = 2$ of them are picked. In the expert icons, a darker color means a larger weight on that stream; row i of a 4×4 mixing expert shows how much output stream i takes from each stream.

Animation 4 MoC: each token is routed to different convolution, deconvolution and mixing experts



Step 9 / 9 · token "linked" · combine and plug in

Every token assembles its own read, write and mixing this way from the expert bank. The experts themselves are trained model parameters; the token only decides which ones to use and how much of each.

Route	$s = \mathrm{Router}(x_l) \qquad R = \mathrm{TopK}(s, k), g = \mathrm{softmax}(s_R)$
Read	$u = \mathrm{Conv}(v; \Sigma_{e \in R_{\mathrm{pre}}} g_e K^{(e)}_{\mathrm{pre}})$
Write	$\Delta v = \mathrm{Deconv}(F(u); \Sigma_{e \in R_{\mathrm{post}}} g_e K^{(e)}_{\mathrm{post}})$
Mix	$m = \mathrm{Mix}(v; \Sigma_{e \in R_{\mathrm{res}}} g_e M^{(e)}) \qquad v_{l+1} = m + \Delta v$

Compared with mHC

	mHC	MoC
Where the read, write and mix kernels come from	Generated directly per token by projections	Routed from three expert groups and combined with weights
What the token decides	The kernel values themselves	Which experts to use, and how much weight each gets
Learned parameters	The projections that generate the kernels, plus static biases	The expert kernels themselves, and the router
Sinkhorn normalization	Once per token at every sublayer	On the expert parameters, shared by all tokens; can be precomputed after training

4. Revisiting the connections mHC actually learns

The analysis cited in the introduction measures the connections that mHC actually produces at all 86 sublayers of DeepSeek-V4-Flash (43 layers, each with an attention and an FFN sublayer) and probes them with inference-time interventions. Viewed through MoC, each of its observations lines up with one of MoC's design choices:

Observation in DeepSeek-V4-Flash	In MoC
Reading and writing concentrate on a few streams. The effective number of streams averages about 2.0 for reading and 1.8 for writing; within an input sequence, 87% (read) and 90% (write) of tokens share the same dominant stream on average; the dominant stream changes with depth.	Read and write experts are fixed connection patterns in their own right, such as "read mainly from streams 1 and 2". Since each sublayer has its own expert bank, different depths can learn different patterns.
Each token needs only a few routes. Keeping only the 3 largest read (or write) weights of every token at every sublayer, rescaled to keep their sum, raises perplexity by only 2.4% (or 2.7%); keeping 2 raises it by 12–14%.	Each token combines only k experts from the bank instead of receiving a fresh set of arbitrary coefficients. The pruning acts on stream weights whereas MoC selects experts, so the correspondence is not exact, but both point the same way: a token needs only a few choices.
Mixing is set mostly by the sublayer; per-token variation matters little. Fixing the mixing matrix of every sublayer in the first 22 layers to its average over C4 samples raises perplexity by only 0.2%.	Mixing experts are static parameters of the sublayer, so the sublayer-specific structure is stored in the experts and the token only chooses among a few of them. With a single mixing expert per sublayer, the mixing matrix stops depending on the token, which is exactly the "fix it to the average" intervention.
Residual mixing is weak in deep layers. The mixing matrices in the last 21 layers are close to the identity overall; replacing all of them with the identity raises perplexity by only 1.9%.	The identity is just one pattern a mixing expert can learn; which layers mix and which pass the streams through unchanged is left for each layer's expert bank to learn.

In short, the connections of the trained DeepSeek-V4-Flash are already close to "fairly fixed patterns per sublayer, plus a little per-token choice". MoC is built around exactly this structure: modules carry the patterns, and the router makes

the choice.

These observations come from a single trained mHC model, and the connections MoC learns show the same structure: routing statistics from a 512-step short run and from the formal run's checkpoint at step 6,000 both exhibit it (Section 5). Whether it still holds after training completes remains to be checked.

---

5. Benefits of MoC

All numbers below are my own measurements. Apart from the formal run, every test ran on a single GPU of the same type. The model has 1.36B parameters and 23 layers, with the same code and configuration as the formal run: MoC uses 4 experts per group ($E = 4$) and picks 2 per token ($k = 2$), with read, write and mix routed separately. The mHC and plain-residual baselines share the same code.

No Sinkhorn at inference

MoC's mixing matrices are expert parameters and do not depend on the token. After training, Sinkhorn runs once per expert and the results are cached; at inference the model only computes the routing weights and a weighted sum of the two selected 4×4 matrices. mHC instead computes its mixing matrices per token, so under the configuration of the mHC paper every token runs 20 Sinkhorn iterations at every sublayer during inference.

The table below reports the time of a single forward pass (inference mode, bf16, median of 30 runs). In the uncompiled rows, the number in parentheses is the time spent in the connections alone, summed over the 46 sublayers.

Input (rows × length)	Plain residual	mHC	MoC (cached expert matrices)	MoC vs. mHC
Without compilation				
1 × 256	15.7 ms	110.3 ms (93.2)	34.7 ms (17.2)	3.2×
1 × 2048	27.7 ms	113.2 ms (93.5)	52.5 ms (25.4)	2.2×
8 × 2048	176.9 ms	390.7 ms (218.5)	345.1 ms (173.3)	1.13×
Compiled with `torch.compile`				
1 × 256	6.8 ms	26.8 ms	18.1 ms	1.48×
1 × 2048	20.6 ms	43.2 ms	41.2 ms	1.05×
8 × 2048	127.4 ms	259.8 ms	263.3 ms	0.99×
First compile	about 25 s	about 4.7 min	about 55 s	

Without compilation, MoC's advantage is largest, and it grows as inputs get shorter. With few tokens, the 20 Sinkhorn iterations that mHC runs at every sublayer turn into a large number of small kernel launches that dominate the forward pass. Compiling with `torch.compile` fuses the Sinkhorn loop and closes most of the gap: on short inputs (1 × 256) MoC is still about 1.5× faster, while on long inputs and large batches the two are on par. Compilation has its own cost, however: for the same inference forward pass, mHC takes about 4.7 minutes to compile and MoC about 55 seconds.

Caching the expert matrices does not change the model's output; the loss is bit-identical with and without it. Even without caching, MoC runs Sinkhorn only once per forward pass for the experts of all sublayers, which adds only about

3.5 ms without compilation.

These measurements use my PyTorch implementation, without hand-written fused kernels like those in the mHC paper. My code does not yet support KV-cache decoding, so a full forward pass on a short input serves as a proxy for latency-sensitive use. Both multi-stream connections are also clearly slower than the plain residual; most of this comes from the memory traffic of reading and writing several streams, which is precisely what the mHC paper's fused kernels target.

Much faster compilation

The table below measures a single training micro-step: a forward and backward pass over 8 × 2048 tokens on one GPU, without the optimizer update, median of 20 runs. Compile times are cold starts from an empty compile cache, with both models compiled side by side under identical conditions.

	mHC	MoC
Uncompiled: time per step	1.086 s	1.003 s
Uncompiled: peak memory	55.1 GiB	52.9 GiB
First compile	40.4 min	2.5 min
Compiled: time per step	0.866 s	0.808 s
Compiled: peak memory	46.7 GiB	46.7 GiB

Once compiled, the two models run at similar speed and memory; the real difference lies in compilation itself, which takes 2.5 minutes for MoC and 40.4 minutes for mHC, about 16 times longer. Since every change to the code or model configuration triggers a recompile, this cost is paid again and again. Compiled MoC is already used in the formal run and has been stable throughout, with a median MFU of about 51% by theoretical FLOPs between steps 1,350 and 6,340. To be fair, the mHC paper optimizes its per-token Sinkhorn with hand-written fused kernels, and mHC-lite likewise notes that an efficient Sinkhorn depends on highly specialized CUDA kernels. The advantage here is therefore one of engineering convenience, not something mHC cannot achieve.

Interpretable and checkable

Because every MoC expert has an explicit index, analysis can work at the level of experts: which experts each layer prefers, which kinds of tokens take which patterns, and whether each expert satisfies its constraints. Analyzing mHC, by contrast, means working with the per-token coefficient matrices directly, as reference 4 does. I ran both kinds of analysis on a 512-step short run and on the formal run's checkpoint at step 6,000, counting routing over 41.9M and 8.4M dev-shard tokens respectively:

Groups of 4 experts, sorted by usage	Step 512	Step 6,000
Share of tokens picking the most used expert	98.7%	99.4%
Second most used	83.2%	84.2%
Third and fourth	15.3%, 3.2%	14.3%, 2.4%
Groups with both slots nearly fixed (of 138)	61	63
Groups whose second slot clearly varies with the token	21	19
Experts almost never picked (of 552)	67	85
Mixing experts with a column sum off by more than 0.05 (of 184)	55	81

The routing structure can be read directly off these counts, and it stays stable in the formal run: in each group one expert is selected almost always, while only the second slot varies among the remaining experts from token to token. This matches the observations in Section 4, namely fairly fixed patterns plus a little per-token choice.

Whether the constraints hold can likewise be checked directly on the parameters. Every mixing expert's row sums are exactly 1 (within 10^{-6}), but 20 Sinkhorn iterations do not bring every column sum to 1: at step 6,000, 81 of the 184 mixing experts have a column sum off by more than 0.05, with a median deviation of 0.04. mHC's per-token matrices share this problem (mHC-lite reports single-layer column sums off by up to 100%), but detecting it there requires statistics over the matrices of many tokens.

6. Summary

- **Finding:** the connections between residual streams can themselves be modular models. Once the N streams are interleaved, mHC's read, write and mix are a convolution and a deconvolution with kernel = stride = N, and a sliding N×N map, so each token's connections are a set of kernels.
- **Proposal:** Modular Connections (MoC) replace the kernels generated per token with routable convolution, deconvolution and mixing experts; each token selects a few experts, as in a mixture of experts, and combines them with weights.
- **Evidence and benefits:** in DeepSeek-V4-Flash, the trained mHC has fairly stable connections at every sublayer, and per-token variation matters little; MoC is designed around exactly this structure. MoC needs no Sinkhorn at inference (2–3× faster without compilation, still about 1.5× faster on short inputs once compiled), compiles far faster, and can be inspected expert by expert.

7. Outlook

My ongoing experiments focus on a single question: for MoC's efficiency gains to translate into real adoption, its quality must match that of mHC.

The formal run so far points in that direction. Using the configuration of Section 5 (1.36B parameters, 23 layers), I am training MoC on about 49B tokens (46,875 steps), evaluating validation BPB on the same validation set every 250 steps

and comparing it with mHC's completed formal run at matching steps. As of November 6, 2026, training has reached step 6,360 (about 14%). Since step 2,000, the validation BPB difference between MoC and mHC has stayed between −0.0009 and +0.0017, while MoC has remained 0.013–0.016 below the plain residual baseline. These are early results from a single seed, and this run also uses `torch.compile` and different hardware from the mHC reference run. A final conclusion will have to wait until training completes.

If you have suggestions or are interested in collaborating, please feel free to reach me at selenyuesu@gmail.com.

Citation

If you find this post useful in your work, please cite it as:

```
@misc{su2026modular,
  author = {Su, Yue},
  title  = {Modular Connections: The Connections Between Residual Streams Can Themselves Be Modular Models},
  year   = {2026},
  month  = nov,
  doi    = {10.5281/zenodo.24032654},
  url    = {https://selen-suyue.github.io/modular-connections/},
  note   = {Blog post}
}
```

A PDF version is archived on Zenodo under DOI 10.5281/zenodo.24032654 and can also be downloaded here.

References

1. Zhu et al. Hyper-Connections. arXiv:2409.19606.
2. DeepSeek-AI. mHC: Manifold-Constrained Hyper-Connections. arXiv:2512.24880.
3. Shazeer et al. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538.
4. Zhao et al. How Does mHC Use Its Residual Streams? Selective Routing and Near-Identity Mixing. arXiv:2609.05309.
5. Yang and Gao. mHC-lite: You Don't Need 20 Sinkhorn-Knopp Iterations. arXiv:2601.05732.