HOPE：一个自引用模块

arjun

来自论文《嵌套学习：深度学习架构的错觉》，作者：Ali Behrouz、Meisam Razaviyayn、Peilin Zhong 和 Vahab Mirrokni — Google Research, NeurIPS 2025

要理解架构，我们应该从最初开始。你知道语言模型有什么问题吗？它们有健忘症。不是那种忘记过去的那种。它们记得自己的预训练得很好。这是一种特定的健忘症，称为顺行性遗忘症，在这种遗忘症中，条件之前的长期记忆是完整的，它们体验当前时刻的能力是完美的。但是它们无法将任何东西从短期记忆转移到长期记忆。30 秒后，刚刚发生的事情就消失了。它们每次都以没有通往明天的桥梁，第一次体验现在。

这正是当前 LLMs 发生的情况。

它们有两种类型的知识：

- 持久知识存储在 MLP 模块和投影权重中。从预训练开始
- 仅当 token 处于注意力窗口中时存在的上下文知识

当上下文消失时，这些知识也随之消失。没有机制将信息从临时（注意力）转移到永久（MLP）。

这一认识是本文的起点。

论文提出了三个核心贡献来解决这个问题：

1. 表达性优化器：证明基于梯度的优化器如 Adam 和带有动量的 SGD 实际上是压缩梯度信息的关联记忆模块
2. 自修改学习模块：一种序列模型，通过学习自己的更新算法来学习如何修改自身
3. 连续体记忆系统：一种新的公式，超越了传统的“长期/短期记忆”二元论

一切皆是联想记忆

在构建任何事物之前，我们需要理解神经网络中记忆的实际含义。事实证明，有一个美丽的统一框架：联想记忆。

什么是联想记忆？它是学习事件之间的映射。当你学习一个人的名字时，你正在将他们的脸映射到他们的名字上。键到值。事件到事件。

形式上，我们可以将其写为一个优化问题：

$$\mathcal{M}^* = \arg\min_{\mathcal{M}} \tilde{\mathcal{L}}(\mathcal{M}(\mathbf{K}); \mathbf{V})$$

> *找到最佳记忆函数 $\mathcal{M}$，将键（如人脸）映射到值（如姓名），并使误差最小。*

我们有键 $\mathbf{K}$ ，值 $\mathbf{V}$ ，以及一个将键映射到值的记忆函数 $\mathcal{M}$ 。损失 $\tilde{\mathcal{L}}$ 衡量这种映射的好坏。

我们所知的多数架构都可以重新表述为关联记忆。目标函数的选择和优化过程决定了你将得到什么样的架构。

示例 I：线性注意力

说 $\mathcal{M}$ 只是一个矩阵（一个线性层）。将目标设置为点积相似度：

$$\tilde{\mathcal{L}} = -\langle \mathcal{M}\mathbf{K}, \mathbf{V}^\top \rangle$$

> *衡量相似度：记忆的输出与值越一致，效果越好。*

如果你用梯度下降来优化这个目标，你会得到线性注意力。递归更新变为：

$$\mathcal{M}_t = \mathcal{M}_{t-1} + \mathbf{v}_t \mathbf{k}_t^\top$$

> *每一步，将当前键和值之间的关联添加到记忆中。简单的累积。*

线性注意力机制本质上就是基于点积目标的梯度下降。

示例 2：Delta 规则

现在将目标函数改为 L2 回归损失：

$$\tilde{\mathcal{L}} = \|\mathcal{M}\mathbf{K} - \mathbf{V}\|_2^2$$

> *均方误差：记忆的预测与实际值相差多远？越小越好。*

使用梯度下降优化，你得到 delta 规则：

$$\mathcal{M}_t = \mathcal{M}_{t-1} - \eta_t(\mathcal{M}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top$$

> *根据其预测误差修正记忆。如果错误，则按错误程度成比例调整。*

不同的目标，不同的架构。但底层结构相同。通过梯度下降优化的联想记忆。

从深度学习的角度来看，我们看到的是解决方案，最终的更新规则。但从嵌套学习的角度来看，我们看到的是内部学习过程。我们看到架构内部存在一个优化问题，而我们正通过某种优化器来解决这个问题。

我们可以通过选择目标或优化器来改进架构，这些选择成为我们可以操纵的设计决策。

梯度下降也是一种联想记忆

取一个简单的 MLP 并用梯度下降进行训练。更新规则是：

$$W_{t+1} = W_t - \eta_t \nabla_W \mathcal{L}(W_t; \mathbf{x}_t)$$

> *向误差梯度的反方向移动权重。就像滚下山找到最低点一样。*

使用链式法则：

$$W_{t+1} = W_t - \eta_t \nabla_{y_t} \mathcal{L}(W_t; \mathbf{x}_t) \otimes \mathbf{x}_t$$

> *更新 = 学习率 × 误差信号 × 输入。外积将输入与输出连接起来。*

我们可以将其重写为：

$$W_{t+1} = \arg\min_{W} \langle W\mathbf{x}_t, \nabla_{y_t}\mathcal{L}(W_t;\mathbf{x}_t)\rangle + \frac{1}{2\eta_t}\|W - W_t\|_2^2$$

> *平衡两个目标：跟随梯度 AND 不要从当前权重变化太多。*

我们试图将输入 $\mathbf{x}t$ 映射到梯度（或"误差信号"） $\nabla_{y_t}\mathcal{L}$ ，使用点积相似度来衡量质量。这看起来就像线性注意力！

但有一个关键区别，在线性注意力中，键和值与记忆状态 $\mathcal{M}$ 无关。你可以预先计算它们。但在这里，值即梯度依赖于当前状态 W_t 。

记忆会生成它自己的学习目标。这被称为自指模型。该模型通过生成自己的值来控制自己的学习过程。

自指学习

$$W_{t+1} = W_t + \eta_{t+1}\mathbf{v}_t \otimes \mathbf{x}_t$$

> *通过在自生成的目标与输入之间添加连接来更新权重。*

其中

$$\mathbf{v}_t = \mathbf{f}_{W_t}(\mathbf{x}_t) = -\nabla_{y_t}\mathcal{L}(W_t;\mathbf{x}_t)$$

> *记忆体生成自己的学习目标！v 是模型认为下一个应该学习的内容。*

在每一步中， $\mathbf{v}_t$ 都是由记忆 W_t 自身生成的。记忆根据它当前所在的位置来决定下一步要学习什么。这比简单的线性注意力机制要强大得多，在简单的线性注意力机制中，你只是将预先给定的键映射到预先给定的值。

梯度下降是一种联想记忆，但它是一种自我指涉的联想记忆。它不是在学习如何映射，而是在学习如何生成用于映射的正确事物，然后学习这种映射。

动量：梯度记忆

给梯度下降添加动量：

$$W_{t+1} = W_t + \mathbf{m}_{t+1}$$

> *权重通过动量量改变，累积过去的梯度信息。*

$$\mathbf{m}_{t+1} = \alpha_{t+1}\mathbf{m}_t - \eta_{t+1}\nabla_W \mathcal{L}(W_t; \mathbf{x}_{t+1})$$

> *动量 = （保留部分上一次动量） + （添加新的梯度）。就像一个具有惯性的滚动的球。*

动量在做什么？它是梯度的一种关联记忆。

到目前为止，所描述的记忆都是针对令牌的，它们将输入令牌映射到输出值。但动量作用于一个不同的上下文，即梯度。它将过去的梯度压缩到其参数中。

并且我们可以将动量表述为解决它自己的优化问题：

$$\min_{\mathbf{m}} -\langle \mathbf{m}, \nabla_W \mathcal{L}(W_t; \mathbf{x}_t) \rangle$$

> *动量本身也在解决一个优化问题：将过去的梯度压缩到一个有用的方向上。*

使用梯度下降进行优化，你将得到动量更新。

嵌套优化问题

当你训练神经网络时，架构是一个优化问题（将标记映射到输出的关联记忆）。优化器也是一个优化问题（将梯度映射到权重更新）

它们是同一回事，只是作用于不同的上下文。架构作用于标记。优化器作用于梯度，它们是相互关联的。架构为优化器生成上下文。优化器看到的梯度？这些来自架构。所以你不能独立设计它们，它们是同一个系统的一部分。

> 这*是嵌套学习：将机器学习模型视为一组嵌套（或并行）的优化问题，每个问题都有其自身的上下文流程。*

这种相互关联具有实际影响。如果你为一个组件选择了一个糟糕的优化问题，它可能会与其他组件相矛盾，导致整个设计崩溃。不同层级可能会相互冲突。理解深度学习架构意味着将一切视为一个相互关联的系统，其中每个组件的上下文都依赖于其他组件。

这也是为什么你不能简单地说“我已经设计了 Adam，让我们用它来训练所有架构。”Transformer 生成的梯度与 MLP 生成的梯度从根本上不同。Transformer 的 Hessian 要复杂得多。优化器需要匹配它所生成的上下文架构。

隐藏维度，频率

在嵌套学习视角中，每个组件都有一个更新频率，即它变化的频率。

在 Transformer 中，注意力机制每个 token 都会更新。频率 = ∞ （非参数化，每次重新计算）。MLP 模块在预训练期间更新，然后冻结。预训练后频率为 0。

预训练本身就是一种情境学习，只是其情境包含数十亿或数万亿个 token。我们将整个预训练数据集压缩到持久组件（MLPs、投影）中。然后注意力机制提供了另一种形式的情境学习，但其情境长度要短得多，只有数千个 token。

两个极端情况。中间什么都没有。为什么不能有很多中间状态呢？

但人脑并不这样工作。脑电波在多种频率下发生。伽马波（30-150 赫兹）用于感觉处理，贝塔波（13-30 赫兹）用于主动思考，Theta/德尔塔波（0.5-8 赫兹）用于记忆巩固。

多重时间尺度。多重频率。统一且可复用的结构，结合大脑中的多时间尺度更新，是人类持续学习的关键要素。嵌套学习允许每个组件进行多时间尺度更新，同时揭示出像 Transformer 这样著名的架构实际上是具有不同频率更新的线性层。这就是我们需要构建的。

连续体记忆系统：遗忘的频谱

与其只有一个 MLP 模块（频率 0），不如让多个 MLP 模块以不同的频率更新：

$$\mathbf{y}_t = \text{MLP}^{(f_k)}(\text{MLP}^{(f_{k-1})}(\cdots \text{MLP}^{(f_1)}(\mathbf{x}_t)))$$

> *链式 MLP 模块，每个模块以不同的速度更新。快速→中等→慢速，类似于脑电波。*

每个 $\text{MLP}^{(f_\ell)}$ 模块拥有自己的块大小 $C^{(\ell)}$ ，该大小决定了其更新频率。以下是关键更新规则：

$$\theta_{i+1}^{(f_\ell)} = \begin{cases} \theta_i^{(f_\ell)} - \sum_{t=i-C^{(\ell)}}^{i} \eta_t^{(\ell)} f(\theta_t^{(f_\ell)}; \mathbf{x}_t) & \text{if } i \equiv 0 \pmod{C^{(\ell)}} \\ 0 & \text{otherwise} \end{cases}$$

> *参数仅在块边界处更新。快速内存每 512 个标记更新一次，慢速内存每 8192 个标记更新一次。*

让我来解读一下这段话的意思。在级别 ℓ 上的参数 $\theta^{(f_\ell)}$ 仅在 i 能被块大小 $C^{(\ell)}$ 整除时更新。当这种情况发生时，它们会累积过去 $C^{(\ell)}$ 个 token 的所有梯度（或者更一般地说，误差信号 $f(\theta_t^{(f_\ell)}; \mathbf{x}_t)$ ），并一次性应用它们。否则，更新为零，参数冻结。

这会给你带来什么？假设你有三个级别：

- 级别 1： $C^{(1)} = 512$ 个 token（高频，适应快，遗忘快）
- 第二级： $C^{(2)} = 2048$ 个 token（中等频率）
- 第三级： $C^{(3)} = 8192$ 个 token（低频率，最持久）

当你处理一个长序列时；第一级每 512 个 token 更新一次。它能看到局部模式，适应快，但遗忘也快。第二级每 2048 个 token 更新一次。它在一个更长的窗口中积累信息，更稳定，反应更慢。第三级每 8192 个 token 更新一次。它捕捉到变化最慢的模式，最持久的知识。

512 个 token 之前学到的内容可能被第一级遗忘（因为它更新并继续了），但第二级仍然保留。2K 个 token 之前的内容可能从第二级消失，但第三级仍然保留。

并且由于所有层级在训练过程中通过反向传播相互连接，知识可以循环回溯。第 3 层级可以提醒第 1 层级它所遗忘的内容。第 2 层级可以在第 1 层级信息消失前巩固其信息。这通过时间维度形成了一个循环。

频率 $f_\ell = \frac{L_{\max}}{C^{(\ell)}}$ 决定了在最大序列长度内一个层级更新的次数。高频率意味着多次更新（局部、自适应）。低频率意味着较少更新（全局、持久）。并且通过堆叠多个具有各自 $C^{(\ell)}$ 的层级，你创建了一个频谱，即连续记忆系统。

这是一种在多个时间尺度上优雅退化的记忆，能够从更新较慢的层级中恢复被遗忘的信息。不是两个极端（冻结的 MLP 和转瞬即逝的注意力），而是一系列巩固过程，就像大脑通过不同的频段将信息从工作记忆转移到中间存储再到长期记忆一样。

Delta 梯度下降：自适应遗忘

标准梯度下降假设数据样本是独立的。但序列中的 token 是相关的。

标准梯度下降：

$$W_{t+1} = \arg\min_{W} \langle W\mathbf{x}_t, \mathbf{u}_t \rangle + \frac{1}{2\eta_t} \|W - W_t\|_2^2$$

> *标准梯度下降：沿着梯度方向移动，同时保持靠近当前权重。*

在 $\mathbf{u}_t = -\nabla_{y_t} \mathcal{L}(W_t; \mathbf{x}_t)$ 处。

> *u 是负梯度，是减少误差的方向。*

用 L2 回归替换点积：

$$W_{t+1} = \arg\min_{W} \frac{1}{2} \|W\mathbf{x}_t - \mathbf{u}_t\|_2^2 + \frac{1}{2\eta_t} \|W - W_t\|_2^2$$

> *回归版本：在不过度偏离当前权重的情况下，最小化预测误差。*

取梯度，设为零（假设已归一化 $\|\mathbf{x}_t\|_2 = \lambda$）：

$$2(W_{t+1}\mathbf{x}_t - \mathbf{u}_t)\mathbf{x}_t^\top + 2\eta_t(W_{t+1} - W_t) = 0$$

> *将梯度设为零以找到最优更新。标准的微积分优化。*

$$W_{t+1}(\mathbf{x}_t\mathbf{x}_t^\top + \eta_t I) = \mathbf{u}_t\mathbf{x}_t^\top + \eta_t W_t$$

> *变形形式：W_new 乘以某个量 = 目标。需要求逆以求解。*

使用 Sherman-Morrison 求逆：

$$(\mathbf{x}_t\mathbf{x}_t^\top + \eta_t I)^{-1} = \frac{1}{\eta_t}\left(I - \frac{1}{\lambda^2 + \eta_t}\mathbf{x}_t\mathbf{x}_t^\top\right)$$

> *矩阵求逆快捷方法（Sherman-Morrison）。避免昂贵的计算。*

回代：

$$W_{t+1} = W_t\left(I - \frac{1}{\lambda^2 + \eta_t}\mathbf{x}_t\mathbf{x}_t^\top\right) - \beta_t \nabla_{y_t}\mathcal{L}(W_t; \mathbf{x}_t)\mathbf{x}_t^\top$$

> *Delta GD：根据输入相似性自适应地遗忘。重复输入 → 忘记更多。*

在 $\beta_t = \frac{1}{\eta_t} - \frac{\lambda^2}{\eta_t(\lambda^2+\eta_t)}$ 处。

看第一个项： $W_t(I - \alpha_t\mathbf{x}_t\mathbf{x}_t^\top)$ 在 $\alpha_t = \frac{1}{\lambda^2+\eta_t}$ 处。

这是一种基于当前输入的自适应衰减！当你反复看到相似的输入时， $\mathbf{x}_t\mathbf{x}_t^\top$ 的值会很大，你衰减得更强，遗忘得更激进。当输入多样化时，你衰减得较少。

> 这*是 Delta 梯度下降（DGD）。记忆会根据数据流的统计信息选择性地遗忘。*

自我修改的巨人：学习如何学习

现在我们准备好构建 HOPE 的核心。但是，巨人是什么？

当你选择 L2 回归作为目标，并使用带有动量和权重衰减的梯度下降进行优化时，巨人就会出现。这就是配方：特定的损失函数和特定的优化器，你就能得到巨人架构。

HOPE 将 Titans 扩展为自指模型。我们不希望假设键和值是给定的，我们希望生成自己的值并从中学习。而且我们希望每个组件都是自适应的：学习率、衰减因子，所有的一切都是由模型自身生成的。

序列模型的标准配方：

$$\mathbf{k}_t = \mathbf{x}_t W_k, \quad \mathbf{v}_t = \mathbf{x}_t W_v, \quad \mathbf{q}_t = \mathbf{x}_t W_q$$

> *标准投影：将输入转换为键（用于匹配的内容）、值（用于返回的内容）、查询（用于查找的内容）。*

$$\min_{\mathcal{M}} \mathcal{L}(\mathcal{M}; \mathbf{k}_t, \mathbf{v}_t)$$

> *找到最佳内存，将键映射到值。*

$$\mathbf{y}_t = \mathcal{M}_t \mathbf{q}_t$$

> *输出 = 将记忆应用于查询。'记忆对这个查询说了什么？'*

预训练的固定投影矩阵 W_k, W_v, W_q ，但我们希望它们能够适应

使每个投影成为一个在上下文中更新的记忆模块：

$$\mathbf{k}_t = \mathcal{M}_{k,t-1}(\mathbf{x}_t), \quad \mathbf{v}_t = \mathcal{M}_{v,t-1}(\mathbf{x}_t)$$

> *键和值现在来自自适应记忆，它们会根据上下文变化！*

我们也想要自适应学习率和衰减因子：

$$\eta_t = \mathcal{M}_{\eta,t-1}(\mathbf{x}_t), \quad \alpha_t = \mathcal{M}_{\alpha,t-1}(\mathbf{x}_t)$$

> *学习速率和衰减现在由模型本身生成。它决定学习速度。*

生成你自己的值。不要从预给定的值中学习，而是从当前的内存状态生成它们

$$\hat{\mathbf{v}}_{\square,t} = \mathcal{M}_{\square,t-1}(\mathbf{v}_t) \quad \text{for } \square \in \{k, v, q, \eta, \alpha, \text{memory}\}$$

> *每个组件生成它自己的学习目标。模型教自己要学习什么。*

每个内存 $\mathcal{M}_\square$ 是一个 2 层 MLP：

$$\mathcal{M}_\square(\cdot) = (\cdot) + W_{\square,1}\sigma(W_{\square,2}(\cdot))$$

> *2 层残差 MLP：输出 = 输入 + 学习到的转换。简单而强大。*

他们全都使用 Delta 梯度下降和自生成目标进行更新：

$$\mathcal{M}_{\square,t} = \mathcal{M}_{\square,t-1}\left(\alpha_t I - \eta_t \mathbf{k}_t \mathbf{k}_t^\top\right) - \eta_t \nabla_{\mathcal{M}_{\square,t-1}} \mathcal{L}(\mathcal{M}_{\square,t-1}; \mathbf{k}_t, \hat{\mathbf{v}}_{\square,t})$$

> *完全自适应更新：衰减旧记忆 + 从自生成目标中学习。所有内容都会自适应。*

每个组件都是自适应的：

- 学习率 η_t ？ 由模型生成
- 衰减因子 α_t ？ 由模型生成
- 数值 $\hat{\mathbf{v}}_{\square,t}$ ？ 由模型生成
- 更新规则？使用 Delta GD 的自适应衰减

这是一个自指的 Titan。模型学习如何修改自身。它学习自己的学习过程。

HOPE：一个具有连续记忆的自指模块

HOPE 的完整架构：

步骤 1：自我修改的泰坦

$$\mathbf{k}_t = \mathcal{M}_{k,C\times\lceil t/C\rceil}(\mathbf{x}_t), \quad \mathbf{v}_t = \mathcal{M}_{v,C\times\lceil t/C\rceil}(\mathbf{x}_t)$$

> *自适应内存中的键/值，在块边界处更新以实现高效的并行处理。*

$$\eta_t = \mathcal{M}_{\eta, C \times \lceil t/C \rceil}(\mathbf{x}_t), \quad \alpha_t = \mathcal{M}_{\alpha, C \times \lceil t/C \rceil}(\mathbf{x}_t)$$

> *每个块生成学习率和衰减。模型控制自己的学习速度。*

$$\hat{\mathbf{v}}_{\square,t} = \mathcal{M}_{\square, C \times \lceil t/C \rceil}(\mathbf{v}_t)$$

> *在块边界处生成自定目标。模型创建自己的课程。*

$$\mathcal{M}_{\square,t} = \mathcal{M}_{\square,t-1}\left(\alpha_t I - \eta_t \mathbf{k}_t \mathbf{k}_t^\top\right) - \eta_t \nabla \mathcal{L}_{\mathcal{M}_{\square, C \times \lceil t/C \rceil}}$$

> *记忆更新：自适应遗忘+梯度步进。平衡保持和学习的平衡。*

$$\mathbf{o}_t = \mathcal{M}_{\text{memory},t}(\mathbf{q}_t)$$

> *自修改泰坦的最终输出：查询更新后的记忆。*

($C \times \lceil t/C \rceil$ 表示我们在块边界处更新内存状态以实现并行化)

步骤 2：连续体记忆系统

$$\mathbf{y}_t = \text{MLP}^{(f_k)}(\text{MLP}^{(f_{k-1})}(\cdots \text{MLP}^{(f_1)}(\mathbf{o}_t)))$$

> *连续体记忆级联：输出流经不同更新频率的多层感知机。*

每个多层感知机级别使用 Delta GD 以自己的频率进行更新。

信息流：

输入 → 自我修改的 Titan（自适应投影、自我生成的目标、学习的学习率）

Titan 输出 → 多级内存（不同的时间尺度）

知识持续存在，可以通过反向传播循环回来

M_3：优化器的连续记忆

如果我们能为架构提供连续记忆，为什么不能为优化器提供呢？

见解：在 Adam 中，我们拥有一阶动量和二阶动量。但 Adam 能否在其动量项中考虑越来越长的信息？这似乎对理解全局损失结构并不太有效，动量仅捕捉局部结构。

因此，我们添加了更新频率不同的额外动量项：

$$\mathbf{m}_{t+1}^{(1)} = \beta_1 \mathbf{m}_t^{(1)} + (1-\beta_1)\nabla_W \mathcal{L}$$

> *一阶动量：快速更新，捕捉局部梯度结构。每步更新。*

$$\mathbf{m}_{i+1}^{(K)} = \begin{cases} \beta_K \mathbf{m}_i^{(K)} + (1-\beta_K)\sum_j \nabla_W \mathcal{L}_j & \text{if } i \equiv 0 \pmod{K} \\ \mathbf{m}_i^{(K)} & \text{otherwise} \end{cases}$$

> *第 K 阶动量：慢速更新，捕捉全局结构。每 K 步更新。*

第一个动量 $\mathbf{m}^{(1)}$ 负责局部结构，它在每一步更新。第二个动量 $\mathbf{m}^{(K)}$ 负责更全局的理解，它只每 K 步更新。然后我们对最终更新进行加权组合。

这为什么重要呢？想象一个持续学习的模型在复杂的损失函数中导航。如果你有弱的记忆管理能力，你只能考虑局部梯度。动量项无法理解全局视角。但拥有更强大的梯度记忆能力时，优化器可以理解损失函数的全局特性，并找到更有效的解决方案。

论文中有一个美丽的例子；一个波动剧烈的损失函数。简单的梯度下降（甚至包括动量）需要花费很长时间才能收敛，因为它不断探索相同的区域。但具有更好内存管理的优化器会记住“我过去已经经过这个点，我不需要从这个方向继续，我可以从那个方向继续。”它正在从自己的优化历史中学习。

自我修改发生在这些记忆生成它们自己的学习目标时。它们不再被告知“将这个键映射到那个值”，而是决定“根据我现在所在的位置，我应该学习下一个内容。”记忆控制它自己的学习过程。

它们都是相同的基本结构。字面意义上。差异仅仅是三个问题：

什么是上下文？是标记还是梯度或其他什么？

频率是多少？你多久更新一次？每个 token？每 512 个 token？从不？

学习规则是什么？梯度下降？Delta 规则？自我指代生成？

那是整个设计空间。

为什么这种架构有效

多层次的适应（高阶情境内学习）

自我修改的 Titan 具有嵌套的优化问题。

常规情境学习："我根据示例调整我的输出。"

HOPE 实现高阶情境学习："我调整我处理示例的方式，我调整我调整的速度，我调整我认为值得调整的内容。"

每个层级都从下一层级学习。

穿越时间的循环

当第一层忘记某些内容时，第二层仍然记得。当第二层忘记时，第三层记得。反向传播让知识可以向后流动：第一层可以查询第二层，第二层可以查询第三层。

自生成课程

在序列早期，Titan 生成"更简单"的目标。随着其完善，它为自己生成更难的目标。隐式课程学习。

自适应遗忘

Delta GD 使遗忘context-dependent. 看到重复模式？更激进地遗忘。看到多样化模式？更多保留。

嵌套学习视角

让我们退一步，看清全局。

嵌套学习揭示的内容几乎是荒谬地简单：一切皆是联想记忆。神经网络的每个组件，无论我们称之为"架构"、"优化器"还是"学习规则"，都只是在某个频率上对某个上下文进行操作的联想记忆。

架构组件是用于标记的联想记忆。注意力操作在频率∞。它每次都从头重新计算，不存储任何持久信息。传统的冻结 MLP 模块操作在频率 0，它们在预训练后永远不会更新。这是两个极端。

优化器是梯度关联存储器。动量压缩过去的梯度。Adam 压缩梯度统计信息。它们在解决与架构相同类型的优化问题，只是在不同的上下文流上。架构看到的是 token；优化器看到的是梯度。

而且还有一个有趣的启示：为什么不能用普通的梯度下降来训练 Transformer 呢？从关联记忆的角度来看，梯度下降是优化器中最简单的关联记忆。当梯度非常复杂时，它无法正确地压缩梯度，而 Transformer 生成的梯度就是复杂的。最近有研究表明，Transformer 生成的梯度的 Hessian 比 MLP 模块等要复杂得多。

Adam 之所以有效，是因为它是一种更强大的关联记忆。在论文中，他们展示了 Adam 是关联记忆对回归损失的优化解。它对梯度的记忆管理更好。当你的架构生成复杂的梯度景观时，你需要一个具有足够内存容量的优化器来有效地压缩这些信息。

连续记忆系统填补了这些极端之间的所有内容。多个 MLP 模块以 512 个 token、2048 个 token、8192 个 token 的频率进行更新。一个从 0 到∞的频谱。每一个仍然是关联记忆，只是在自己的时间尺度上运行。

让一切豁然开朗的直觉

传统深度学习说：“垂直堆叠层以获得深度。让它们更深更宽以获得更多参数。”

嵌套学习说：“在不同时间尺度上堆叠优化过程。让它们以不同频率更新以获得持续的内存巩固。”

这里就是实际的好处：我们在一个领域学到的所有知识都可以迁移到另一个领域。所有关于长上下文建模的讨论，都直接适用于优化器。优化器的上下文是梯度，如果你的优化器内存管理较弱，它只能看到局部梯度，无法理解全局损失函数的形状。

想想看：一个持续学习器需要在一个复杂的损失函数形状中导航。对于梯度来说，如果优化器只有短上下文记忆，它就会短视。而如果优化器有长上下文记忆，它就能识别"我之前来过这里"，并做出更好的决策。那些扩展 token 上下文长度的技术，同样可以扩展梯度上下文长度。

当你用 Adam 优化器训练 Transformer 时，你并没有"一个被优化的神经网络"。你拥有嵌套的记忆：Adam 的动量（梯度快速记忆，每步更新）、注意力权重（token 无限频率记忆，每次前向传播重新计算）和 MLP 模块（token 零频率记忆，训练后冻结）。

HOPE 只是将这一点明确化并填补了空白。它说："如果我们有每 512、2048、8192 个 token 更新记忆呢？如果这些记忆能够生成自己的学习目标呢？如果每个组件都能调整自己的学习率呢？"

深度学习架构的错觉在于认为这些是根本不同的事物。注意力机制与多层感知机与优化器。嵌套学习的现实是认识到它们都是同一事物：关联记忆以不同频率压缩它们的上下文流。

Transformer 为我们提供了两种频率。HOPE 为我们提供了一个频谱。这就是从短期记忆到长期记忆的桥梁。不是单一机制，而是一系列以不同速度运行的记忆，每个记忆都从前一个记忆中学习，每个记忆都能够提醒其他记忆他们所遗忘的内容。

这就是神经网络如何解决顺行性遗忘的方法。不是通过构建单一完美的记忆，而是通过构建一个能够自然地在时间尺度上整合信息的记忆层次结构。

持续学习智能能发展到多远？

这离真正的持续学习有多近？诚实的答案是还有许多探索空间。

这是其中一个方向，并且它与其他方法正交。我们需要更强大的优化器来更好地管理内存。我们需要稀疏内存机制。我们需要从多个方面理解持续学习。

但嵌套学习的美妙之处在于它是可组合的。更好的优化器可以接入这个框架。稀疏内存技术可以与之结合。关于长上下文架构所学的知识适用于优化器。关于优化器动量的所学知识适用于架构内存。

前进的道路并非寻找一个万能药。而是构建一套正交技术工具箱，这些技术可以组合：嵌套学习、连续记忆系统、自指模块、更好的优化器内存、稀疏检索，以及从理解架构和优化器之间深层统一性中涌现的任何其他技术。

我们还没有达到涅槃。但我们已经建起了一座桥梁，这是将信息从短暂转移到持久的第一个真正机制。而这正是长期以来一直破损的部分。