APPROXIMATING FUNCTION SPACE DISTANCE FOR CONTINUAL LEARNING IN TRANSFORMERS

Nikita Dhawan
University of Toronto, Vector Institute
nikita@cs.toronto.edu

Felix Dangel
Vector Institute
felix.dangel@vectorinstitute.ai

Roger Grosse
University of Toronto, Vector Institute, Schwartz Reisman institute
rgrosse@cs.toronto.edu

ABSTRACT

Measuring how neural network functions evolve during training, finetuning, or editing is critical for several applications. Such shifts can be formalized via a function space distance (FSD) — the expected squared difference in network outputs under a data distribution — but computing the true FSD requires dataset access that is often infeasible. The previously proposed Linearized Activation Function TRick (LAFTR) circumvents this challenge via specific approximations for linear networks with ReLU activations. We extend this to a more general LInearized Function TRick (LIFTR) to enable data-free FSD estimation for arbitrary architectures, with particular focus on transformers. Our approach decomposes FSD estimation into moment propagation using only pre-computed activation statistics of the data, resulting in a modular implementation that easily generalizes to arbitrary functions. On a modular arithmetic continual learning task, we show that a stochastic variant of LIFTR approaches oracle performance while outperforming parameter-space linearization baselines. LIFTR estimates correlate strongly with oracle FSD and produce better-aligned gradients than competing methods. We further demonstrate that LIFTR degrades more gracefully with network depth than global parameter-space linearization. Code implementations are available here.

1 INTRODUCTION

Quantifying shifts in the functions represented by neural networks is becoming more useful and more challenging as model and dataset sizes continue to grow. When a pretrained model undergoes continual training (Ke et al., 2023), finetuning (Yosinski et al., 2014), or editing (Sinitsin et al., 2020), we are often interested in measuring or even preserving functional relationships previously learned on some data distribution. At large scales, it becomes infeasible to store all these relationships naïvely, due to computational, storage, privacy, and/or proprietary restrictions. These is especially relevant for transformer models (Vaswani et al., 2017) for which performance continues to improve with scale (Kaplan et al., 2020) but even open-sourced model weights are rarely accompanied by a complete release of their training datasets (Zhang et al., 2022; Touvron et al., 2023).

Function space distances (FSDs). FSDs capture changes in functional relationships learned by a neural network $F(x, \boldsymbol{\theta})$ under a *parameter drift* $\boldsymbol{\theta}_0 \to \boldsymbol{\theta}_1$. Here, we define FSD as the expected squared Euclidean distance between network outputs under a given data distribution, $D(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, p_{\text{data}}) \coloneqq \mathbb{E}_{x \sim p_{\text{data}}}[\|F(x; \boldsymbol{\theta}_1) - F(x; \boldsymbol{\theta}_0)\|^2]$. Unlike parameter-space metrics, this FSD captures functional discrepancies on actual inputs, which is critical where functional consistency matters more than weight similarity (Benjamin et al., 2018). Evaluating the empirical "oracle" FSD requires a forward pass on an entire dataset, averaging the squared norms of the induced output layer's *representation drift* $\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1) \coloneqq F(x; \boldsymbol{\theta}_1) - F(x; \boldsymbol{\theta}_0)$. The computational, memory, and access constraints discussed above make this strong oracle prohibitively expensive in practice. Linearized approximations of the network can, however, significantly improve efficiency at the expense of some performance drop. Examples include global parameter linearization (Jacot et al., 2018; Lee et al.,

2019) that uses a coreset of datapoints or layer-wise activation linearization (Dhawan et al., 2023) that enables FSD estimation without any datapoints.

Parameter-space linearization. The Neural Tangent Kernel (NTK) (Jacot et al., 2018; Lee et al., 2019) framework can approximate network behavior analytically by linearizing it with respect to its parameters and simplifying FSD computations to Jacobian-vector products. This requires storing data, typically in the form of a coreset of datapoints at which the required Jacobian is computed. Other than its storage costs, such global parameter linearization has the disadvantage of becoming an increasingly coarse estimate as network depth increases. Even for a simple network with multiple linear layers, parameter linearization is inexact. This is because multi-layer networks contain nonlinear relationships between parameters of different layers, which are not captured by globally linearizing the network with respect to its parameters, as opposed to a layer-wise linearization.

Activation linearization and moment propagation. The Linearized Activation Function TRick (LAFTR) from Dhawan et al. (2023) used layer-wise activation linearization for data-free FSD estimation in linear networks with ReLU activations, outperforming global parameter linearization in tasks like continual learning. LAFTR reduces FSD estimation to approximating the moments of *output representation drift*, which is practically achieved by propagating moments of *intermediate representation drifts*. The approximate moment propagation requires aggregate statistics, precomputed once with a single forward pass on the data. Beyond fully-connected and convolutional networks with ReLU activations, it remains unclear how to apply LAFTR to arbitrary architectures, like transformers, that introduce other nonlinear operations (*e.g.* attention) and residual connections. Hence, this work aims to present a more general LInearized Function TRick (LIFTR) to allow data-free FSD estimation for any architecture, with particular focus on, and empirical evaluation of, transformers.

Our contribution. We present a general recipe for layer-wise linearization with respect to a layer's inputs in any network architecture, which reduces to the original LAFTR method for ReLU multilayer perceptrons and convolutional networks. Specifically, we break up FSD estimation into: (i) parameter and activation linearization for each step of a forward pass, (ii) corresponding precomputation of statistics, and (iii) moment propagation of representation drifts, visualized in Figure 1. LIFTR follows a modular recipe applicable to any architecture, but requires implementing a routine for each operation to compute expected-Jacobian-vector products under linear and independence assumptions. We explore both deterministic and stochastic variants, and further approximations for efficiency.

We evaluated LIFTR on a continual learning task with transformer models, using FSD approximations to penalize drift from earlier task solutions. Our method variants achieved performance close to that of the oracle, with the stochastic variant outperforming baselines that rely on global parameter-space linearization. Further experimental analysis showed that our method's FSD estimates correlate strongly with the oracle FSD in magnitude as well as gradient cosine similarity. Finally, we investigated the effect of network depth on different FSD estimators' quality and performance and again, observed advantages of layer-wise linearization over global parameter-space linearization.

The key contributions and findings of this work are:

1. We generalize the LAFTR FSD estimator, and its variants, to any neural network architecture and describe a generic LInearized Function TRick (LIFTR) for data-free FSD estimation.
2. We implement our method for transformers and empirically evaluate its variants on a modular arithmetic continual learning task. LIFTR variants outperform other baselines while storing only aggregate statistics instead of actual datapoints.
3. LIFTR variants more closely estimate the oracle FSD and its gradients, and suffer more gradual decline in performance with increasing network depth than other linearization-based estimators.

2 PRELIMINARIES

Let $F(x; \boldsymbol{\theta})$ be the function computed by a neural network on input x using parameters $\boldsymbol{\theta}$. Given two sets of parameters $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_1$, the expected Euclidean Function Space Distance (FSD) between them with respect to an input distribution p_{data} is given by

$$D(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, p_{\text{data}}) = \mathbb{E}_{x \sim p_{\text{data}}}[\|\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)\|^2] \coloneqq \mathbb{E}_{x \sim p_{\text{data}}}[\|F(x; \boldsymbol{\theta}_1) - F(x; \boldsymbol{\theta}_0)\|^2]. \quad (1)$$

Consider the motivating example of continual learning (Wang et al., 2023; Normandin et al., 2021) of a sequence of tasks $t \in \{0, \ldots, T\}$, using loss function $\mathcal{L}$ and a penalty on the FSD between current

parameters $\boldsymbol{\theta}_t$ and parameters $\{\boldsymbol{\theta}_i\}_{i=0}^{t-1}$ fit to previous tasks. FSDs are computed over the previously seen data distributions $\{p_i\}_{i=0}^{t-1}$ and scaled by a hyperparameter λ. For $T = 2$, we optimize:

$$\min_{\boldsymbol{\theta}_1} \mathcal{L}(\boldsymbol{\theta}_1) + \lambda D(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, p_0). \tag{2}$$

Motivated by such settings, we assume for any FSD that we have access to the previously trained $\boldsymbol{\theta}_0$ and current parameters $\boldsymbol{\theta}_1$, but it is too expensive or infeasible to store an entire previously seen dataset of N inputs to approximate Equation (1) with a Monte Carlo estimate, $\frac{1}{N}\sum_{n=1}^{N}\|\Delta F(x_n; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)\|^2$. This empirical FSD is an oracle estimate, upper-bounding the accuracy of any other estimator.

Several practical approximations to Equation (1) can be derived from linearizing the network with respect to parameters $\boldsymbol{\theta}_0$, *i.e.* $F^{\text{lin}}_{\theta_0} = F(x, \boldsymbol{\theta}_0) + J_{\boldsymbol{\theta}} F|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$, using the network's Jacobian $J_{\boldsymbol{\theta}} F$. This reduces the FSD to a quadratic form of $\Delta\boldsymbol{\theta} := \boldsymbol{\theta}_1 - \boldsymbol{\theta}_0$, and the Gauss-Newton matrix (Schraudolph, 2002), for which several efficient approximations exist. These may rely on a coreset of datapoints to locally estimate the Jacobian or make independence and diagonal assumptions to estimate the FSD parametrically. We refer the reader to Section 2 of Dhawan et al. (2023) and related literature in Section 4 for more discussion on parameter-space linearization.

Following Dhawan et al. (2023), we note that $\mathbb{E}[\|\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)\|^2] = \|\mathbb{E}[\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)]\|^2 + tr(\text{Cov}(\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)))$ and that the FSD computation is reduced to the computation of the first two moments of $\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)$. Further, assume that F decomposes as a sequence of m computations, or *steps*, $f^{(i)}$, each parameterized by $\theta^{(i)}$ for $i \in \{1, \ldots, m\}$, such that $\boldsymbol{\theta} = (\theta^{(1)}, \ldots, \theta^{(m)})$. Note that $\theta^{(i)}$ is empty if the i-th step has no parameters, *e.g.* activation functions. We describe a linear approximation to every computation step, $f^{(i)}$, with respect to all its inputs, which will lead to a parametric approximation of the FSD, *i.e.*, one that does not require access to any datapoints.

For ease of notation, we drop the superscript (i) and refer to a single generic computation step as f, with (possibly) multiple inputs $z = (z[1], \ldots, z[k])$, parameters θ, and a single output $y = f(z; \theta)$, as visualized in (i) of Figure 1. Multiple inputs to f imply its dependence on outputs of one or more previous steps, allowing for branching and merging within the computation graph, *e.g.* residual connections or attention mechanisms. Subscripts, such as z_0, y_0 or z_1, y_1 refer to computations obtained using parameters $\boldsymbol{\theta}_0$ or $\boldsymbol{\theta}_1$ respectively. See Appendix A for a list of all notation used here.

3 LINEARIZED FUNCTION TRICK (LIFTR)

3.1 STEP-WISE LINEARIZATION

Consider the approximation $f^{\text{lin}}_{\theta_0, z_0}(z; \theta)$, that linearizes f with respect to all its inputs, centered at (z_0, θ_0), which are the inputs to f in the computation graph parameterized by $\boldsymbol{\theta}_0$. Specifically,

$$f^{\text{lin}}_{\theta_0, z_0}(z; \theta) = f(z_0; \theta_0) + \sum_{j=1}^{k} \left[J_{z[j]}\big|_{z=z_0, \theta=\theta_0} (z[j] - z_0[j]) \right] + J_\theta|_{z=z_0, \theta=\theta_0} (\theta - \theta_0), \tag{3}$$

where $J_a|_{a=a_0}$ is the Jacobian of f with respect to a, evaluated at $a = a_0$. Note that f and its linearization $f^{\text{lin}}_{\theta_0, z_0}$ match at the anchor point, $f(z_0; \theta_0) = f^{\text{lin}}_{\theta_0, z_0}(z_0; \theta_0)$. Since the linearization is always with respect to z_0 and θ_0, we drop these subscripts from now on for ease of presentation.

Applying this approximation to f under some continually trained parameters θ_1, we can estimate the output representation drift $\Delta y = y_1 - y_0$ as $\Delta y^{\text{lin}} = f^{\text{lin}}(z_1^{\text{lin}}; \theta_1) - f^{\text{lin}}(z_0^{\text{lin}}; \theta_0) = f^{\text{lin}}(z_1^{\text{lin}}; \theta_1) - f(z_0; \theta_0)$. For any f, the input z is either the input data x or the output of a previous computation step. In the latter case, z^{lin} denotes the approximation of z, obtained by linearizing the previous computation step. Note that all Jacobians in Equation (3) are evaluated at z_0 and θ_0, so they may be precomputed and stored in any sequential training procedure, before continued training yields θ_1. As depicted in (ii) of Figure 1, we have the following recursion:

$$\Delta y \approx \Delta y^{\text{lin}} = f^{\text{lin}}(z_1^{\text{lin}}; \theta_1) - f^{\text{lin}}(z_0^{\text{lin}}; \theta_0) = \sum_{j=1}^{k} \left[(J_{z[j]}) \Delta z^{\text{lin}}[j] \right] + (J_\theta)\Delta\theta, \tag{4}$$

where $\Delta\theta = \theta_1 - \theta_0$ and $\Delta z^{\text{lin}}[j] = z_1^{\text{lin}}[j] - z_0^{\text{lin}}[j]$ is the representation drift Δy^{lin} from a previous computation step. The base case of this recursion is $\Delta z^{(0)} = \Delta x = 0$ since $z_0^{(0)} = z_1^{(0)} = x$.

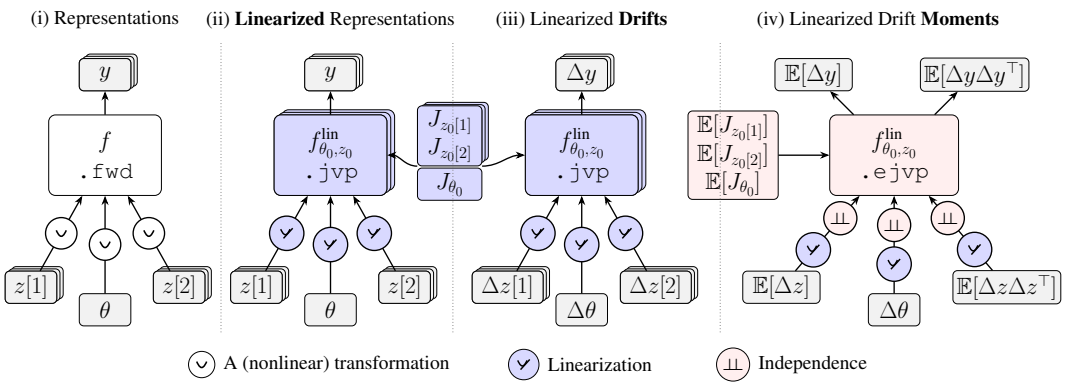

Figure 1: **Illustration of applying LIFTR to a single operation in a neural network.** Given a (possibly nonlinear) function computation f to produce intermediate representations y in (i), LIFTR linearizes f around reference parameters θ_0 and inputs z_0 in (ii), propagates input drift terms Δz through the linearized $f^{\text{lin}}_{\theta_0,z_0}$ to obtain linearized representation drift Δy in (iii), and finally assumes independence between Jacobians and drifts to propagate linearized drift moments via expected-Jacobian-vector products, or `ejvp`s, to obtain $\mathbb{E}[\Delta y]$ and $\mathbb{E}[\Delta y \Delta y^\top]$ in (iv). Hence, separate forward passes or JVPs per datapoint in (i)-(iii) are aggregated into `eJVP`s in (iv).

3.2 Approximate Moment Propagation

To compute the first two moments of $\Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)$, we assume independence between input drifts and Jacobians and drop cross-covariance terms between different inputs, giving:

$$\begin{aligned}
\mathbb{E}_x[\Delta y^{\text{lin}}] &\approx \sum_{j=1}^{k} \mathbb{E}_x[J_{z[j]}]\mathbb{E}_x[\Delta z^{\text{lin}}[j]] + \mathbb{E}_x[J_\theta]\Delta\theta, \\
\mathbb{E}_x[\Delta y^{\text{lin}} \Delta y^{\text{lin}\top}] &\approx \sum_{j=1}^{k} \mathbb{E}_x[J_{z[j]}]\mathbb{E}_x[\Delta z^{\text{lin}}[j] \Delta z^{\text{lin}}[j]^\top]\mathbb{E}_x[J_{z[j]}]^\top + \mathbb{E}_x[J_\theta]\Delta\theta\Delta\theta^\top\mathbb{E}_x[J_\theta]^\top.
\end{aligned} \tag{5}$$

In words, approximate moment propagation pushes all incoming drift moments through their *expected* Jacobian $\mathbb{E}_x[J_a]$, and then sums them. Since $JCJ^\top = J(JC^\top)^\top$, for some matrix C, requires only left multiplication with J, the propagation only relies on *expected*-Jacobian-vector products, or `eJVP`s: $c \mapsto \mathbb{E}_x[J]c$. Each step of a forward pass requires its own implementation of `eJVP`s.

Finally, to treat the full neural network, we approximate each layer by its linearization and then propagate the first two moments of $\Delta z^{(0)} = 0$ through the network to obtain the first two moments of intermediate representation drifts, as shown in (iv) of Figure 1. Finally, the approximated first two moments of $\Delta y^{(m)} = \Delta F(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)$ yield the approximate Euclidean FSD.

3.3 Step-specific Operations

To support a given step or layer f, we need to implement the following operations:

1. a precomputation stage that aggregates and stores the necessary statistics to compute `eJVP`s, before any continued training, and
2. `eJVP` functions: $c \mapsto \mathbb{E}_x[J_a]c$ for all inputs a to f, that rely on access to the stored statistics.

In practice, statistics can be computed after obtaining $\boldsymbol{\theta}_0$ by iterating over the dataset once to store an online running average over mini-batches. Parameter-free steps have $\Delta\theta = 0$ and so, don't need to store J_θ. Since Jacobians only appear in `eJVP`s, we can exploit the structure of each computation function f to store memory-efficient versions or approximations of the expected Jacobians. For example, a ReLU layer stores the fraction of times each unit is activated as its empirical expected Jacobian and its `eJVP` computes an element-wise product. Since second moments scale quadratically in size, we experimented with replacing each $\mathbb{E}_x[(\Delta y^{\text{lin}})(\Delta y^{\text{lin}})^\top]$ with its diagonal approximation in Section 5. See Table 1 for standard transformer operations, their `eJVP`s, and storage costs.

Function f	**Inputs**	**Output** $y = f(\cdot)$	**eJVPs** $\mathbb{E}[J]\,c$	**Stochastic approx.**	**Storage Cost (Full, Diagonal)**
Linear	$z \in \mathbb{R}^d$ (W, b)	$zW^\top + b$	$cW^\top$ $zv^\top + b$	$z \sim \mathcal{N}(\mu_z, \Sigma_z)$	$O(d^2),\ O(d)$
ReLU	$z \in \mathbb{R}^d$	$\max(0, z)$	$\mu_{f'} \odot c$	$f' \sim \text{Bernoulli}(\mu_{f'})$	$O(d),\ O(d)$
Query-Key Dot Product	$q \in \mathbb{R}^{s\times d}$ $k \in \mathbb{R}^{s\times d}$	$\frac{1}{\sqrt{d}} qk^\top$	$\frac{1}{\sqrt{d}} c\mu_k^\top$ $\frac{1}{\sqrt{d}} \mu_q c^\top$	$q \sim \mathcal{N}(\mu_q, I)$ $k \sim \mathcal{N}(\mu_k, I)$	$O(sd),\ O(sd)$ $O(sd),\ O(sd)$
Softmax	$z \in \mathbb{R}^{h\times s\times s}$	$\text{softmax}(z)$	$(\mu_y I - \Sigma_y)\, c$	$y \sim \text{Dirichlet}(\cdot)$	$O(hs^3),\ O(hs^2)$
Weighted Values	$a \in \mathbb{R}^{h\times s\times s}$ $v \in \mathbb{R}^{s\times d}$	av	$\mu_a c$ $c\mu_v$	$y \sim \text{Dirichlet}(\cdot)$ $v \sim \mathcal{N}(\mu_v, I)$	$O(sd),\ O(sd)$ $O(hs^3),\ O(hs^2)$
Add (Skip Connections)	a b	$a + b$	c c	—	$0, 0$ $0, 0$
Causal Mask	z	$\text{tril}(\mathbf{1}) \odot z$	$\text{tril}(c)$	—	$0, 0$

Table 1: Summary of deterministic expected-Jacobian operators and stochastic approximations used to propagate moments through standard transformer computation steps. Storage costs per step are for the full or diagonal second moments. $\mu_\bullet$ and $\Sigma_\bullet$ denote precomputed first two moments under p_{data}.

3.4 Stochastic variant

Instead of deterministically computing and storing the first two moments of Δy^{lin} for each step, as described above, it is also possible to implement a stochastic version of the method, that propagates *samples* of Δy^{lin} through each step. We do so by imposing distributions over the quantities required to compute the eJVPs using the stored statistics and drawing S samples $\{\Delta y_s^{\text{lin}}\}_{s=1}^S$ from them. This eventually yields S samples $\{\Delta F_s^{\text{lin}}(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)\}_{s=1}^S$ and a stochastic estimate of the FSD is given by $\frac{1}{S}\sum_{s=1}^S \|\Delta F_s^{\text{lin}}(x; \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)\|^2$. This stochastic version of our method can be computationally more efficient since it avoids computing covariances of the output differences. Empirically, we also find that it can improve performance in continual learning, which often benefits from stochasticity. We refer to this stochastic version as LIFTR-S, as opposed to the previously described deterministic version, which we call LIFTR-D. Diagonal approximations are indicated with "-Diag".

4 Related Work

Network Linearization. Parameter-space linearization has been used as a general tool to study neural network training trajectories (Jacot et al., 2018; Lee et al., 2019). Elastic Weight Consolidation (EWC, Kirkpatrick et al., 2017) is a canonical continual learning method that applies a diagonal approximation on top of network linearization with respect to parameters. More recently, Nam et al. (2025) argued for layer-wise linear models as simple yet useful models to understand neural network dynamics and Afzal et al. (2025) showed that first-order parameter-space linearization can explain fine-tuning dynamics in large language models. Recently, Porrello et al. (2026) developed a data-free FSD estimator for such linear fine-tuning settings using connections to curvature matrix approximations. For task arithmetic (Ilharco et al., 2022), fine-tuning linearized models reportedly reduces cross-task inference between task vectors (Ortiz-Jimenez et al., 2023). Parameter linearization, or the NTK framework, can also fail in practice, since standard training dynamics (Fort et al., 2020) and performance (Chizat et al., 2019) of commonly used networks often diverge from their parameter-space linearization. Activation-space linearization is less common in the literature, but has been considered by Dhawan et al. (2023) for ReLU MLPs and convolutional networks and by Erdogan et al. (2025) for attention layers in large language models.

Transformer Function Space Distance. Function space distance approximations for transformers have several use-cases that have been considered in the existing literature. Model editing methods (De Cao et al., 2021; Meng et al., 2022) aim to update language models with new information and commonly use a regularizer similar to a function space distance to preserve previously learned knowledge. Continual learning (Ermis et al., 2022; Pelosin et al., 2022; Huang et al., 2021; Ding et al., 2022) or continued fine-tuning (Wang et al., 2026) methods explicitly prevent catastrophic forgetting by regularizing drift in network outputs for previously learned tasks.

Moment propagation. Propagation of moments of intermediate representations has also found applications in the literature. Wright et al. (2024) present moment propagation for uncertainty

quantification. Noci et al. (2022) analyze signal propagation in transformers and relatedly, Kedia et al. (2024) derive closed-form expressions for the first and second-order moments of the outputs and gradients of each component of a transformer to understand several optimization issues.

5 Empirical Evaluation

We designed our experiments to answer the following questions:

1. How well do LIFTR-based regularizers prevent catastrophic forgetting in continual learning, compared to existing baselines (Section 5.1)?
2. How accurately do different FSD estimators approximate the magnitude and gradients of the oracle FSD between transformers (Section 5.2)?
3. Since layer-wise linearization and independence assumptions accumulate approximation errors per layer, what is the effect of depth on FSD estimators' performance (Section 5.3)?

Experimental Setup. We evaluated several FSD approximation methods on a continual learning task that requires a transformer model to solve modular addition and subtraction sequentially, inspired by Nanda et al. (2023). Specifically, inputs are length-3 sequences: $[a, \texttt{fn-token}, b]$ and targets are given by $\texttt{fn}(a, b)$ modulo p, where $p = 113$, $a, b \in \{0, \ldots, p-1\}$, $\texttt{fn} \in \{\texttt{add}, \texttt{sub}\}$ is the arithmetic operation, and `fn-token` represents a special token corresponding to the operation `fn`. We trained a transformer with two attention blocks, embedding dimension $d_{\text{model}} = 512$, $h = 4$ attention heads, and feedforward hidden dimension $d_{\text{hidden}} = 1024$, for 200 epochs, using cross-entropy loss, AdamW optimizer, learning rate of $3 \cdot 10^{-4}$, and minibatch size of 256.

We first trained the transformer on the addition task, using data from p_{add} to obtain $\boldsymbol{\theta}_{\text{add}}$, and then on subtraction, using data from p_{sub} to obtain $\boldsymbol{\theta}_{\text{sub}}$, which risks loss of performance on p_{add}. To prevent this, we added a regularization term given by the FSD $D(\boldsymbol{\theta}_{\text{add}}, \boldsymbol{\theta}_{\text{sub}}, p_{\text{add}})$ between previously learned parameters $\boldsymbol{\theta}_{\text{add}}$ and current parameters $\boldsymbol{\theta}_{\text{sub}}$ on previous task data from p_{add}, weighted by a scalar hyperparameter λ. We tuned λ via grid-search on a validation dataset to optimize for the final average accuracy across both tasks. LIFTR requires us to propagate the real-valued moments of the integer-valued input data through the network. Since it unclear how to propagate real-valued moments through a standard embedding layer, we treated the input embeddings as the input data and propagated the embedding moments. This is made feasible by freezing the embedding parameters after the first task such that its input embeddings do not change after continued training.

5.1 Continual Learning Performance

We evaluated the impact of different FSD approximations on continual learning performance in the arithmetic task described above based on two common metrics: (i) final average accuracy across tasks, given by $\frac{1}{T}\sum_{i=1}^{T} A_{T,i}$, where $A_{T,i}$ is the accuracy on task i after training on $T = 2$ tasks, and (ii) average backward transfer (BWT) across tasks, given by $\frac{1}{T-1}\sum_{i=1}^{T-1}(A_{T,i} - A_{i,i})$, where $A_{i,i}$ is the accuracy on task i immediately after training on it. BWT directly measures forgetting, with more negative values indicating greater forgetting.

Table 2: Evaluated on an arithmetic continual learning task, LIFTR-S and its diagonal approximation (indicated with -Diag) achieve average accuracy and backward transfer close to the oracle upper bound, outperforming all other baselines.

FSD Estimator	Avg Acc (%)	Avg BWT
Oracle	99.40 ± 0.49	1.05 ± 0.73
LIFTR-S	**98.16 ± 0.45**	-1.43 ± 0.09
LIFTR-S-Diag	98.01 ± 0.35	**-1.05 ± 0.48**
LIFTR-D	94.94 ± 0.31	-9.15 ± 0.22
LIFTR-D-Diag	87.89 ± 0.87	-21.23 ± 0.52
NTK-32	94.24 ± 0.72	-9.71 ± 1.86
RandomSubset-32	54.83 ± 0.20	-87.93 ± 1.38
EWC	51.22 ± 0.62	-95.58 ± 1.23

Table 2 shows these metrics for different FSD estimators with the best performing λ, including the oracle, which stores the entire training set to compute the FSD on a different minibatch at every iteration. LIFTR-S and its diagonal version (indicated by -Diag) are close to this oracle performance, outperforming all other baselines as well as the deterministic method, LIFTR-D. Other baselines include the nonparametric NTK-32 which linearizes the network with respect to its parameters using a coreset of 32 datapoints,

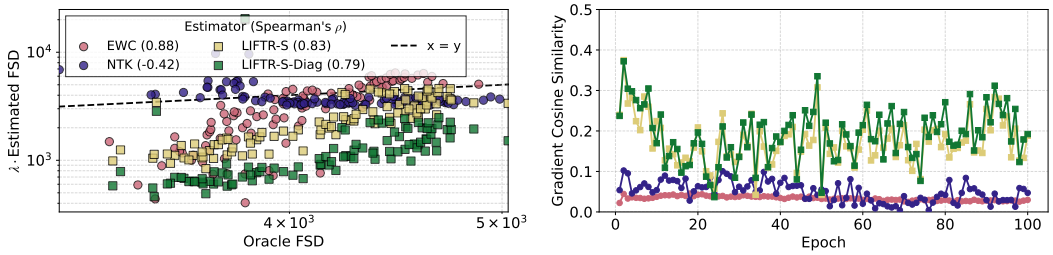

Figure 3: LIFTR-S and EWC approximations were closely correlated in magnitude with the oracle FSD (**Left**). For ease of visualization, FSD estimates for each method were scaled by its own λ, selected to minimize the mean squared error between the estimated and oracle FSDs. LIFTR-S and LIFTR-S-Diag also produced gradients with greater cosine similarity to those of the oracle FSD (**Right**), as compared to all other methods.

RandomSubset-32, which computes FSD on the same coreset of 32 randomly selected datapoints at each iteration of training, and the parametric EWC which make an additional diagonal approximation to the NTK method.

In the continual learning objective (see Equation (2)), λ controls the trade-off between preventing forgetting of the previous task and plasticity (Dohare et al., 2024) of the network to learn a new task. It can be more informative to compare estimators using their Pareto fronts of previous task accuracy versus current task accuracy for a range of values of λ. We include this result in Figure 2, where each point represents a continual training run of 100 epochs per task for a given FSD estimator and a given value of λ in $[10^{-5}, 5\cdot10^{-5}, 10^{-4}, 5\cdot10^{-4}, 10^{-3}, 5\cdot10^{-3}, 10^{-2}, 5\cdot10^{-2}, 0.1, 0.5, 1, 5, 10]$. Increasing marker sizes indicate increasing values of λ. Intuitively, as λ increases, FSD regularization improves accuracy on the previously learned task (0), while reducing performance on the current task (1). Greater area under the curve corresponds to a more desirable trade-off between the two tasks. The oracle achieved greatest area under the curve with LIFTR-S and LIFTR-S-Diag outperforming all other methods.

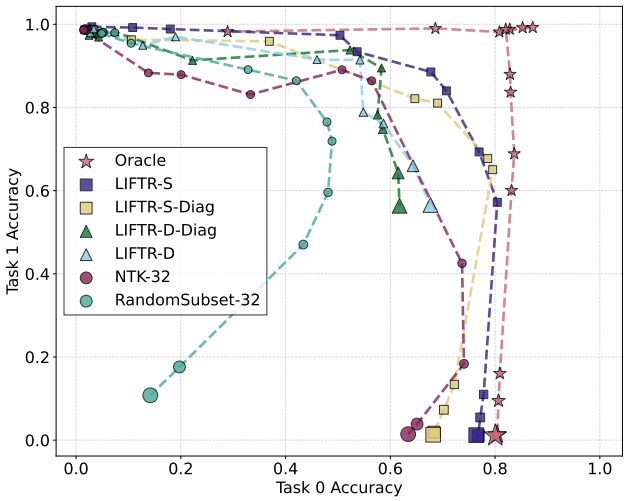

Figure 2: LIFTR-S and LIFTR-S-Diag achieved greater areas under their Pareto front curves than other methods, exhibiting a superior trade-off between preventing forgetting of a previous task and plasticity to learn a new task.

5.2 FSD Approximation Quality

We used the sequential training setup above to measure and compare different FSD estimates against the oracle FSD computed using the entire dataset. After obtaining trained parameters $\boldsymbol{\theta}_{\text{add}}$ on the initial addition task, we continually trained the same network on the subtraction task with no FSD regularization such that as training continues, the network is expected to move further from $\boldsymbol{\theta}_{\text{add}}$ in parameter space as well as function space. At each epoch, we computed the oracle FSD and the approximate FSD estimated by the best performing methods, LIFTR-S and LIFTR-S-Diag, and linearization-based baselines, NTK and EWC. Figure 3 (Left) shows that stochastic LIFTR and EWC estimates of the FSD correlate strongly with the oracle FSD, as measured by the Spearman's rank correlation coefficient (Spearman, 1987), reported in parentheses.

The high magnitude correlation of EWC estimates but poor continual learning performance seem contradictory but may be explained by investigating the gradients produced by them. In the context of

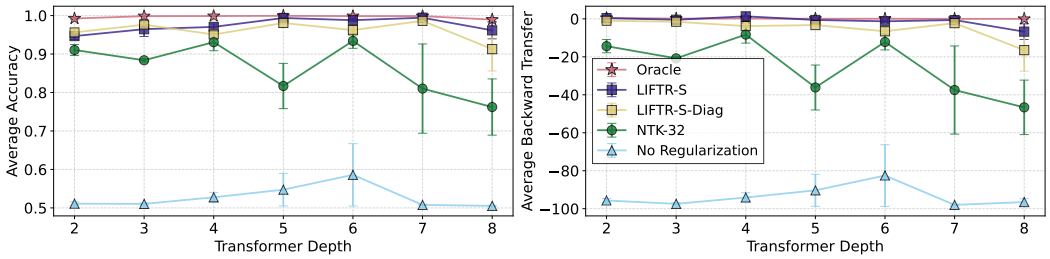

Figure 4: With increasing depth, average accuracy (**Left**) first improved, likely due to greater network capacity, before decreasing, while average backward transfer (**Right**) steadily declined. LIFTR-S methods outperformed NTK overall in terms of both metrics, with slower decline with depth.

continual learning, FSD approximations are used to augment the loss function and are minimized with a gradient-based optimizer, making the gradients produced by them a better heuristic for performance than their magnitude. Hence, with the same setup as before, we computed the cosine similarity between gradients produced by the oracle FSD and those of different FSD estimators at each epoch of continued training. As shown in Figure 3 (Right), we found that the gradients produced by LIFTR-S and LIFTR-S-Diag have greater cosine similarity to those produced by the oracle FSD than other methods. Notably, while EWC produced FSD magnitudes that were strongly correlated with the oracle, the gradient similarity between the two was very small, explaining its poor continual learning performance. In contrast, the stochastic LIFTR estimates aligned well with the oracle in terms of both, magnitude correlation as well as gradient cosine similarity.

5.3 Effect of Depth on Linearization

Our methods and several baselines make a linear approximation to one or another computation in the network as well as independence assumptions to enable efficiency and data-free computations. Since these errors accumulate with each layer, the quality of the approximation is expected to degrade with the depth of a network. Hence, we investigated the change in continual learning performance using the LIFTR-S and NTK methods with increasing depth of the sequentially trained transformer model. Figure 4 shows that average accuracy (Left) and average BWT (Right) on the continual arithmetic task was upper-bounded by the Oracle and lower-bounded by sequential training with no FSD regularization, as expected. Average accuracy initially increased with depth, presumably due to the increasing capacity of the network, before decreasing at larger depths. For all estimators, average BWT decreased with depth, likely because adding more layers accumulated greater errors in FSD estimation, which in turn led to more forgetting. LIFTR methods generally outperfomed NTK, with slower decline in performance as the number of layers in the transformer increased. This is intuitive and similar to the findings in Dhawan et al. (2023) for other architectures, since LIFTR makes step-wise linear approximations and captures more nonlinear interactions between network parameters than NTK, which linearizes the network outputs with respect to all its parameters.

6 Conclusions and Future Work

This work generalized the Linearized Activation Function TRick to the LInearized Function TRick (LIFTR) to enable data-free function space distance (FSD) estimation for arbitrary architectures. We demonstrated how to implement and compose expected-Jacobian-vector products for standard operations to obtain a moment propagation scheme that stores only aggregate statistics and does not require access to actual datapoints. We proposed deterministic and stochastic variants of LIFTR, each with its more memory-efficient diagonal approximation. LIFTR achieved continual learning performance close to an oracle upper bound on an arithmetic task with transformers, which may be explained by the greater alignment of its gradients with those of the oracle.

Future work includes exploring other efficient approximations beyond diagonalization to better understand the fidelity-efficiency tradeoff of the method and its variants. It would be practically useful to explore alternatives to handling integer-valued data other than freezing the embeddings after initial training. Also of great practical interest is validation at larger scales to enable, for instance, continual training and adaptation of large language models without access to their training data.

REFERENCES

Zahra Rahimi Afzal, Tara Esmaeilbeig, Mojtaba Soltanalian, and Mesrob I Ohannessian. Linearization explains fine-tuning in large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Ari S Benjamin, David Rolnick, and Konrad Kording. Measuring and regularizing networks in function space. *arXiv preprint arXiv:1805.08289*, 2018.

Lenaic Chizat, Edouard Oyallon, and Francis Bach. On lazy training in differentiable programming. *Advances in neural information processing systems*, 32, 2019.

Nicola De Cao, Wilker Aziz, and Ivan Titov. Editing factual knowledge in language models. *arXiv preprint arXiv:2104.08164*, 2021.

Nikita Dhawan, Sicong Huang, Juhan Bae, and Roger Baker Grosse. Efficient parametric approximations of neural network function space distance. In *International Conference on Machine Learning*, pp. 7795–7812. PMLR, 2023.

Yuxuan Ding, Lingqiao Liu, Chunna Tian, Jingyuan Yang, and Haoxuan Ding. Don't stop learning: Towards continual learning for the clip model. *arXiv preprint arXiv:2207.09248*, 2022.

Shibhansh Dohare, J Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A Rupam Mahmood, and Richard S Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024.

Mete Erdogan, Francesco Tonin, and Volkan Cevher. Efficient large language model inference with neural block linearization. *arXiv preprint arXiv:2505.21077*, 2025.

Beyza Ermis, Giovanni Zappella, Martin Wistuba, Aditya Rawal, and Cedric Archambeau. Memory efficient continual learning with transformers. *Advances in Neural Information Processing Systems*, 35:10629–10642, 2022.

Stanislav Fort, Gintare Karolina Dziugaite, Mansheej Paul, Sepideh Kharaghani, Daniel M Roy, and Surya Ganguli. Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the neural tangent kernel. *Advances in Neural Information Processing Systems*, 33:5850–5861, 2020.

Yufan Huang, Yanzhe Zhang, Jiaao Chen, Xuezhi Wang, and Diyi Yang. Continual learning for text classification with information disentanglement based regularization. *arXiv preprint arXiv:2104.05489*, 2021.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089*, 2022.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Zixuan Ke, Yijia Shao, Haowei Lin, Tatsuya Konishi, Gyuhak Kim, and Bing Liu. Continual pre-training of language models. *arXiv preprint arXiv:2302.03241*, 2023.

Akhil Kedia, Mohd Abbas Zaidi, Sushil Khyalia, Jungho Jung, Harshith Goka, and Haejun Lee. Transformers get stable: An end-to-end signal propagation theory for language models. *arXiv preprint arXiv:2403.09635*, 2024.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114 (13):3521–3526, 2017.

Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *Advances in neural information processing systems*, 32, 2019.

Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer. *arXiv preprint arXiv:2210.07229*, 2022.

Yoonsoo Nam, Seok Hyeong Lee, Clementine CJ Domine, Yeachan Park, Charles London, Wonyl Choi, Niclas Goring, and Seungjai Lee. Position: Solve layerwise linear models first to understand neural dynamical phenomena (neural collapse, emergence, lazy/rich regime, and grokking). *arXiv preprint arXiv:2502.21009*, 2025.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

Lorenzo Noci, Sotiris Anagnostidis, Luca Biggio, Antonio Orvieto, Sidak Pal Singh, and Aurelien Lucchi. Signal propagation in transformers: Theoretical perspectives and the role of rank collapse. *Advances in Neural Information Processing Systems*, 35:27198–27211, 2022.

Fabrice Normandin, Florian Golemo, Oleksiy Ostapenko, Pau Rodriguez, Matthew D Riemer, Julio Hurtado, Khimya Khetarpal, Ryan Lindeborg, Lucas Cecchi, Timothée Lesort, et al. Sequoia: A software framework to unify continual learning research. *arXiv preprint arXiv:2108.01005*, 2021.

Guillermo Ortiz-Jimenez, Alessandro Favero, and Pascal Frossard. Task arithmetic in the tangent space: Improved editing of pre-trained models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Francesco Pelosin, Saurav Jha, Andrea Torsello, Bogdan Raducanu, and Joost van de Weijer. Towards exemplar-free continual learning in vision transformers: an account of attention, functional and weight regularization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3820–3829, 2022.

Angelo Porrello, Pietro Buzzega, Felix Dangel, Thomas Sommariva, Riccardo Salami, Lorenzo Bonicelli, and Simone Calderara. Dataless weight disentanglement in task arithmetic via kronecker-factored approximate curvature. In *International Conference on Learning Representations (ICLR)*, 2026.

Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 2002.

Anton Sinitsin, Vsevolod Plokhotnyuk, Dmitriy Pyrkin, Sergei Popov, and Artem Babenko. Editable neural networks. *arXiv preprint arXiv:2004.00345*, 2020.

Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 100(3/4):441–471, 1987.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wenqing Wang, Da Li, Xiatian Zhu, and Josef Kittler. mergetune: Continued fine-tuning of vision-language models. *arXiv preprint arXiv:2601.10497*, 2026.

Xiao Wang, Yuansen Zhang, Tianze Chen, Songyang Gao, Senjie Jin, Xianjun Yang, Zhiheng Xi, Rui Zheng, Yicheng Zou, Tao Gui, et al. Trace: A comprehensive benchmark for continual learning in large language models. *arXiv preprint arXiv:2310.06762*, 2023.

Oren Wright, Yorie Nakahira, and José MF Moura. An analytic solution to covariance propagation in neural networks. In *International Conference on Artificial Intelligence and Statistics*, pp. 4087–4095. PMLR, 2024.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

A NOTATION

This section summarizes the notation used throughout the paper.

x	Network input sampled from p_{data}
p_{data}	Input data distribution
$F(x;\boldsymbol{\theta})$	Neural network function mapping input x to output using parameters $\boldsymbol{\theta}$
$\boldsymbol{\theta}$	All network parameters, $\boldsymbol{\theta} = (\theta^{(1)}, \ldots, \theta^{(m)})$
$\boldsymbol{\theta}_0, \boldsymbol{\theta}_1$	Reference and updated parameters
m	Total number of computation steps in the network
$f^{(i)}$	The i-th computation step in the network, where $i \in \{1, \ldots, m\}$
$\theta^{(i)}$	Parameters of the i-th computation step (empty if step has no parameters)
$f(z;\theta)$	Generic computation step with inputs z and parameters θ
$f^{\text{lin}}(z;\theta)$	Linearized approximation of f around reference point (z_0, θ_0)
z	Generic input to a computation step, $z = (z[1], \ldots, z[k])$
k	Number of inputs to a computation step
y	Output of a computation step, $y = f(z;\theta)$
z_0, y_0	Intermediates computed using parameters $\boldsymbol{\theta}_0$
z_1, y_1	Intermediates computed using parameters $\boldsymbol{\theta}_1$
$z^{\text{lin}}, y^{\text{lin}}$	Linearized approximations of intermediates
ΔF	Output difference, $\Delta F(x;\boldsymbol{\theta}_0,\boldsymbol{\theta}_1) = F(x;\boldsymbol{\theta}_1) - F(x;\boldsymbol{\theta}_0)$
$\Delta\theta$	Parameter difference, $\Delta\theta = \theta_1 - \theta_0$
Δy	Output difference for a computation step, $\Delta y = y_1 - y_0$
Δy^{lin}	Linearized approximation of output difference, $\Delta y^{\text{lin}} = y_1^{\text{lin}} - y_0^{\text{lin}}$
$D(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, p_{\text{data}})$	Function Space Distance (FSD), $\mathbb{E}_{x\sim p_{\text{data}}}[\|\Delta F(x;\boldsymbol{\theta}_0,\boldsymbol{\theta}_1)\|^2]$
J_a	Jacobian of f with respect to input a, evaluated at reference point
$\mathbb{E}[J_a]$	Expected Jacobian of f with respect to a over p_{data}
eJVP	Expected-Jacobian-vector product, $c \mapsto \mathbb{E}_x[J_a]c$
$\mathbb{E}[\cdot]$	Expectation over input distribution p_{data}
$\text{Cov}(\cdot)$	Covariance matrix over input distribution p_{data}
$\mu_\bullet, \Sigma_\bullet$	Precomputed first and second moments of quantity $\bullet$
S	Sample size for Monte Carlo estimation in LIFTR-S
ΔF_s^{lin}	The s-th sample of output difference in stochastic approximation