Transformers Learn Higher-Order Optimization Methods for In-Context Learning: A Study with Linear Models

Deqing Fu, Tian-Qi Chen, Robin Jia, Vatsal Sharan
Department of Computer Science
University of Southern California
Los Angeles, CA
{deqingfu,tchen939,robinjia,vsharan}@usc.edu

Abstract

Transformers are remarkably good at *in-context learning* (ICL)—learning from demonstrations without parameter updates—but how they perform ICL remains a mystery. Recent work suggests that Transformers may learn in-context by internally running Gradient Descent, a first-order optimization method. In this paper, we instead demonstrate that Transformers learn to implement higher-order optimization methods to perform ICL. Focusing on in-context linear regression, we show that Transformers learn to implement an algorithm very similar to *Iterative Newton's Method*, a higher-order optimization method, rather than Gradient Descent. Empirically, we show that predictions from successive Transformer layers closely match different iterations of Newton's Method *linearly*, with each middle layer roughly computing 3 iterations. In contrast, *exponentially* more Gradient Descent steps are needed to match an additional Transformers layer; this suggests that Transformers have an comparable rate of convergence with high-order methods such as Iterative Newton, which are exponentially faster than Gradient Descent. We also show that Transformers can learn in-context on ill-conditioned data, a setting where Gradient Descent struggles but Iterative Newton succeeds. Finally, we show theoretical results which support our empirical findings and have a close correspondence with them: we prove that Transformers can implement k iterations of Newton's method with $\mathcal{O}(k)$ layers.

1 Introduction

Transformer neural networks [43] have become the default architecture for natural language processing [12, 6, 27], and have even been adopted by other areas like computer vision [13]. As first demonstrated by GPT-3 [6], Transformers excel at *in-context learning* (ICL)—learning from input-output pairs provided as inputs to the model, without updating their model parameters. Through in-context learning, Transformer-based Large Language Models (LLMs) can achieve state-of-the-art few-shot performance across a wide variety of downstream tasks [31, 37, 42, 9].

Given the importance of Transformers and ICL, many prior efforts have attempted to understand how Transformers perform in-context learning. Prior work suggests Transformers learn classification similar to support vector machines [41, 40] and can approximate linear functions well in-context [14]. Specifically to linear regression tasks, prior work has tried to understand the ICL mechanism and the dominant hypothesis is that Transformers learn in-context by running optimizations internally through gradient-based algorithms [44, 45, 1, 11].

Mathematics of Modern Machine Learning Workshop at NeurIPS 2023.

This paper presents strong evidence for a competing hypothesis: Transformers trained to perform in-context linear regression learn to implement a higher-order optimization method rather than a first-order method like Gradient Descent. In particular, Transformers implement a method very similar to Newton-Schulz's Method, also known as the *Iterative Newton's Method*, which iteratively improves an estimate of the inverse of the design matrix to compute the optimal weight vector. Across many layers of the Transformer, subsequent layers approximately compute more and more iterations of Newton's Method, with increasingly better predictions; both eventually converge to the optimal minimum-norm solution found by ordinary least squares (OLS). Interestingly, this mechanism is specific to Transformers: LSTMs do not learn these same higher-order methods, as their predictions do not even improve across layers.

We present both empirical and theoretical evidence for our claims. Empirically, Transformer induced weights and residuals are similar to Iterative Newton and deeper layers match Newton with more iterations (see Figures 3 and 7). Transformers can also handle ill-conditioned problems without requiring significantly more layers, where GD would suffer from slow convergence but Iterative Newton would not. Crucially, Transformers share the same rate of convergence as Iterative Newton and are exponentially faster than Gradient Descent. Theoretically, we show that Transformer circuits can efficiently implement Iterative Newton, with the number of layers depending linearly on the number of iterations and the dimensionality of the hidden states depending linearly on the dimensionality of the data. Overall, our work provides a mechanistic account of how Transformers perform in-context learning that not only explains model behavior better than previous hypotheses, but also hints at what makes Transformers so well-suited for ICL compared with other neural architectures.

2 Problem Setup

In this paper, we focus on the following linear regression task. The task involves n examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. The examples are generated from the following data generating distribution $P_{\mathcal{D}}$, parameterized by a distribution $\mathcal{D}$ over $(d \times d)$ positive semi-definite matrices. For each sequence of n in-context examples, we first sample a ground-truth weight vector $\boldsymbol{w}^\star \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \in \mathbb{R}^d$ and a matrix $\boldsymbol{\Sigma} \overset{\text{i.i.d.}}{\sim} \mathcal{D}$. For $i \in [n]$, we sample each $\boldsymbol{x}_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$. The label y_i for each $\boldsymbol{x}_i$ is given by $y_i = \boldsymbol{w}^{\star\top} \boldsymbol{x}_i$. Note that for much of our experiments $\mathcal{D}$ is only supported on the identity matrix $\boldsymbol{I} \in \mathbb{R}^{d\times d}$ and hence $\boldsymbol{\Sigma} = \boldsymbol{I}$, but we also consider some distributions over ill-conditioned matrices which will give rise to ill-conditioned regression problems.

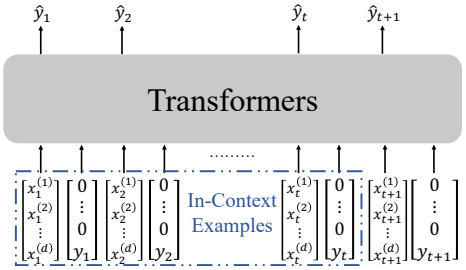


Figure 1: Illustration of how Transformers are trained to do in-context linear regression.

We will use neural network models such as Transformers to solve this linear regression task. As shown in Figure 1, at time step $t+1$, the model sees the first t in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^t$, and then makes predictions for $\boldsymbol{x}_{t+1}$, whose label y_{t+1} is not observed by the Transformers model.

We randomly initialize our models and then train them on the linear regression task to make predictions for every number of in-context examples t, where $t \in [n]$. Training and test data are both drawn from $P_{\mathcal{D}}$. To make the input prompts contain both $\boldsymbol{x}_i$ and y_i, we follow same the setup as Garg et al. [14]'s to zero-pad y_i's, and use the same decoder-only GPT-2 model [30] with softmax activation and causal attention mask (discussed later at definition 1).

We now present the key mathematical details for the Transformer architecture, and how they can be used for in-context learning. First, the causal attention mask enforces that attention heads can only attend to hidden states of previous time steps, and is defined as follows.

Definition 1 (Causal Attention Layer)**.** *An **causal** attention layer with M heads and activation function σ is denoted as* Attn *on any input sequence $\boldsymbol{H} = [\boldsymbol{h}_1, \cdots, \boldsymbol{h}_N] \in \mathbb{R}^{D\times N}$, where D is the dimension of hidden states and N is the sequence length. In the vector form,*

$$\tilde{\boldsymbol{h}}_t = [\text{Attn}(\boldsymbol{H})]_t = \boldsymbol{h}_t + \sum_{m=1}^{M} \sum_{j=1}^{t} \sigma\left(\langle \boldsymbol{Q}_m \boldsymbol{h}_t, \boldsymbol{K}_m \boldsymbol{h}_j \rangle\right) \cdot \boldsymbol{V}_m \boldsymbol{h}_j. \tag{1}$$

Vaswani et al. [43] originally proposed the Transformer architecture with the Softmax activation function for the attention layers. Later works have found that replacing $\mathrm{Softmax}(\cdot)$ with $\frac{1}{t}\mathrm{ReLU}(\cdot)$ does not hurt model performance [7, 36, 48]. The Transformers architecture is defined by putting together attention layers with feed forward layers:

Definition 2 (Transformers)**.** *An L-layer decoder-based transformer with Causal Attention Layers is denoted as $\mathrm{TF}_{\boldsymbol{\theta}}$ and is a composition of a MLP Layer (with a skip connection) and a Causal Attention Layers. For input sequence $\boldsymbol{H}^{(0)}$, the transformers ℓ-th hidden layer is given by*

$$\mathrm{TF}^{\ell}_{\boldsymbol{\theta}}(\boldsymbol{H}^{(0)}) := \boldsymbol{H}^{(\ell)} = \mathrm{MLP}_{\boldsymbol{\theta}^{(\ell)}_{\mathrm{mlp}}}\left(\mathrm{Attn}_{\boldsymbol{\theta}^{(\ell)}_{\mathrm{attn}}}(\boldsymbol{H}^{(\ell-1)})\right) \tag{2}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\theta}^{(\ell)}_{\mathrm{mlp}}, \boldsymbol{\theta}^{(\ell)}_{\mathrm{attn}}\}_{\ell=1}^{L}$ and $\boldsymbol{\theta}^{(\ell)}_{\mathrm{attn}} = \{\boldsymbol{Q}^{(\ell)}_m, \boldsymbol{K}^{(\ell)}_m, \boldsymbol{V}^{(\ell)}_m\}_{m=1}^{M}$ consists of M heads at layer ℓ.

In particular for the linear regression task, Transformers perform in-context learning as follows

Definition 3 (Transformers for Linear Regression)**.** *Given in-context examples $\{\boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_t, y_t\}$, Transformers make predictions on a query example $\boldsymbol{x}_{t+1}$ through a readout layer parameterized as $\boldsymbol{\theta}_{\mathrm{readout}} = \{\boldsymbol{u}, v\}$, and the prediction $\hat{y}^{\mathrm{TF}}_{t+1}$ is given by*

$$\hat{y}^{\mathrm{TF}}_{t+1} := \mathrm{ReadOut}\Big[\underbrace{\mathrm{TF}^{L}_{\boldsymbol{\theta}}(\{\boldsymbol{x}_1, \boldsymbol{y}_1, \cdots, \boldsymbol{x}_t, \boldsymbol{y}_t, \boldsymbol{x}_{t+1}\})}_{\boldsymbol{H}^{(L)}}\Big] = \boldsymbol{u}^{\top}\boldsymbol{H}^{(L)}_{:,2t+1} + v. \tag{3}$$

Our central research question is:

Does the algorithm Transformers learn for linear regression resemble any known algorithm?

We discuss various known algorithms we compare Transformers with in the Appendix A and we claim that *Transformers are more similar to Iterative Newton's Method than Gradient Descent.*

3 Mechanistic Evidence

Our empirical evidence demonstrates that Transformers behave much more similarly to Iterative Newton's than to Gradient Descent. Iterative Newton is a higher-order optimization method, and is algorithmically more involved than Gradient Descent. We begin by first examining this difference in complexity. As discussed in Section 2, the updates for Iterative Newton are of the form,

$$\hat{\boldsymbol{w}}^{\mathrm{Newton}}_{k+1} = \boldsymbol{M}_{k+1}\boldsymbol{X}^{\top}\boldsymbol{y} \quad \text{where} \quad \boldsymbol{M}_{k+1} = 2\boldsymbol{M}_k - \boldsymbol{M}_k\boldsymbol{S}\boldsymbol{M}_k \tag{4}$$

and $\boldsymbol{M}_0 = \alpha\boldsymbol{S}$ for some $\alpha > 0$. We can express $\boldsymbol{M}_k$ in terms of powers of $\boldsymbol{S}$ by expanding iteratively, for example $\boldsymbol{M}_1 = 2\alpha\boldsymbol{S} - 4\alpha^2\boldsymbol{S}^3$, $\boldsymbol{M}_2 = 4\alpha\boldsymbol{S} - 12\alpha^2\boldsymbol{S}^3 + 16\alpha^3\boldsymbol{S}^5 - 16\alpha^4\boldsymbol{S}^7$, and in general $\boldsymbol{M}_k = \sum_{s=1}^{2^{k+1}-1}\beta_s\boldsymbol{S}^s$ for some $\beta_s \in \mathbb{R}$ (see Appendix C.3 for detailed calculations). Note that k steps of Iterative Newton's requires computing $\Omega(2^k)$ moments of $\boldsymbol{S}$. Let us contrast this with Gradient Descent. Gradient Descent updates for linear regression take the form,

$$\hat{\boldsymbol{w}}^{\mathrm{GD}}_{k+1} = \hat{\boldsymbol{w}}^{\mathrm{GD}}_{k} - \eta(\boldsymbol{S}\hat{\boldsymbol{w}}^{\mathrm{GD}}_{k} - \boldsymbol{X}^{\top}\boldsymbol{y}). \tag{5}$$

Like Iterative Newton, we can express $\hat{\boldsymbol{w}}^{\mathrm{GD}}_{k}$ in terms of powers of $\boldsymbol{S}$ and $\boldsymbol{X}^{\top}\boldsymbol{y}$. However, after k steps of Gradient Descent, the highest power of $\boldsymbol{S}$ is only $O(k)$. This exponential separation is consistent with the exponential gap in terms of the parameter dependence in the convergence rate—$\mathcal{O}\left(\kappa(\boldsymbol{S})\log(1/\epsilon)\right)$ steps for Gradient Descent compared to $\mathcal{O}(\log\kappa(\boldsymbol{S}) + \log\log(1/\epsilon))$ steps for Iterative Newton. Therefore, a natural question is whether Transformers can actually represent as complicated of a method as Iterative Newton with only polynomially many layers?

Theorem 1 shows that this is indeed possible.

Theorem 1. *There exist Transformer weights such that on any set of in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^{n}$ and test point $\boldsymbol{x}_{\mathrm{test}}$, the Transformer predicts on $\boldsymbol{x}_{\mathrm{test}}$ using $\boldsymbol{x}^{\top}_{\mathrm{test}}\hat{\boldsymbol{w}}^{\mathrm{Newton}}_{k}$. Here $\hat{\boldsymbol{w}}^{\mathrm{Newton}}_{k}$ are the Iterative Newton updates given by $\hat{\boldsymbol{w}}^{\mathrm{Newton}}_{k} = \boldsymbol{M}_k\boldsymbol{X}^{\top}\boldsymbol{y}$ where $\boldsymbol{M}_j$ is updated as*

$$\boldsymbol{M}_j = 2\boldsymbol{M}_{j-1} - \boldsymbol{M}_{j-1}\boldsymbol{S}\boldsymbol{M}_{j-1}, 1 \le j \le k, \quad \boldsymbol{M}_0 = \alpha\boldsymbol{S},$$

for some $\alpha > 0$ and $\boldsymbol{S} = \boldsymbol{X}^{\top}\boldsymbol{X}$. The number of layers of the transformer is $\mathcal{O}(k)$ and the dimensionality of the hidden layers is $\mathcal{O}(d)$.

We note that our proof uses full attention instead of causal attention and ReLU activations for the self-attention layers. The definitions of these and the full proof appear in Appendix C.

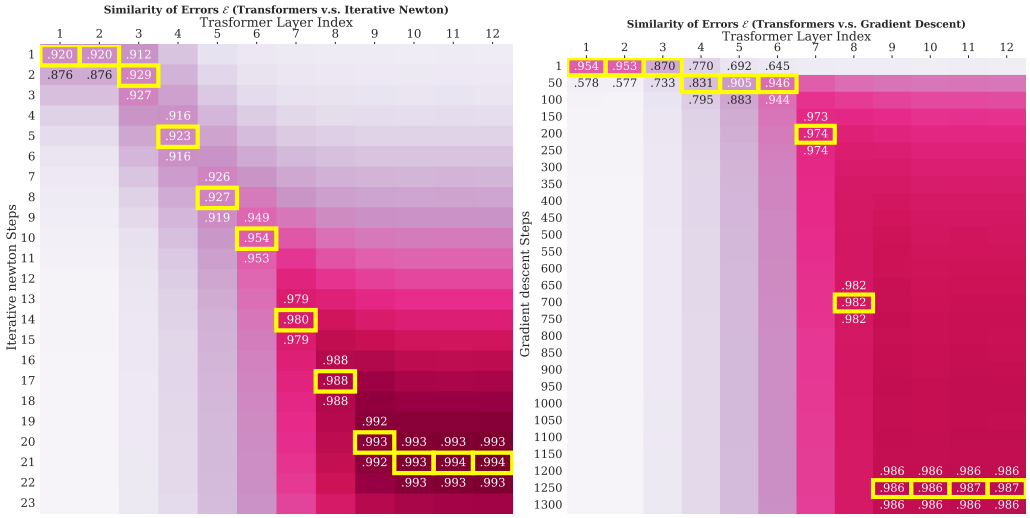


Figure 2: **Heatmaps of Similarity.** The best matching hyper-parameters are highlighted in yellow. See Figure 5 for an additional heatmap where Gradient Descent's steps are shown in log scale.

4 Experimental Evidence

We mainly work on the Transformers-based GPT-2 model with 12 layers with 8 heads per layer. Alternative configurations with fewer heads per layer also support our findings and we defer them to Appendix B. We initially focus on isotropic cases where $\mathbf{\Sigma} = \boldsymbol{I}$ and later consider ill-conditioned $\mathbf{\Sigma}$ in §B.3. Our training setup is exactly the same as [14]: models are trained with at most $n = 40$ in-context examples for $d = 20$ (with the same learning rate, batch size etc.).

We test the more specific hypothesis that the iterative updates performed across Transformer layers are similar to the iterative updates for known iterative algorithms. Specifically, we test whether each layer ℓ of the Transformer corresponds to performing k steps of some iterative algorithm, for some k depending on ℓ. We focus here on Gradient Descent and Iterative Newton's Method; we will discuss online algorithms in Appendix B.6.

For each layer ℓ of the Transformer, we measure the best-matching similarity (see Def. 4) with candidate iterative algorithms with the optimal choice of the number of steps k. As shown in Figure 2, the Transformer has very high error similarity with Iterative Newton's method at all layers. Moreover, we see a clear *linear* trend between layer 3 and layer 9 of the Transformer, where each layer appears to compute roughly 3 additional iterations of the Iterative Newton's method. This trend only stops at the last few layers because both algorithms converge to the OLS solution; Newton is known to converge to OLS (see §A.1), and we verify in Appendix B.2 that the last few layers of the Transformer also basically compute OLS (see Figure 8 in the Appendix). We observe the same trends when using similarity of induced weights as our similarity metric (see Figure 6 in the Appendix),

In contrast, even though GD has a comparable similarity with the Transformers at later layers, their best matching follows an *exponential* trend. For well-conditioned problems, to achieve ϵ error, the convergence rate of GD is $\mathcal{O}(\log(1/\epsilon))$ while the convergence rate of Newton is $\mathcal{O}(\log\log(1/\epsilon))$ (see §A.1). Therefore the convergence rate of Newton is exponentially faster than GD. Transformer's *linear* correspondence with Newton and its *exponential* correspondence with GD provides strong evidence that the rate of convergence of Transformers is similar to Newton, i.e., $\mathcal{O}(\log\log(1/\epsilon))$.

Overall, we conclude that a Transformer trained to perform in-context linear regression learns to implement an algorithm that is very similar to Iterative Newton's method, not Gradient Descent. Starting at layer 3, subsequent layers of the Transformer compute more and more iterations of Iterative Newton's method. This algorithm successfully solves the linear regression problem, as it converges to the optimal OLS solution in the final layers.

References

[1] Kwangjun Ahn, Xiang Cheng, Hadi Daneshmand, and Suvrit Sra. Transformers learn to implement preconditioned gradient descent for in-context learning. *ArXiv*, abs/2306.00297, 2023. 1, 23

[2] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. *ArXiv*, abs/2211.15661, 2022. 10, 18, 19, 23

[3] Yu Bai, Fan Chen, Haiquan Wang, Caiming Xiong, and Song Mei. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. *ArXiv*, abs/2306.04637, 2023. 24

[4] Adi Ben-Israel. An iterative method for computing the generalized inverse of an arbitrary matrix. *Mathematics of Computation*, 19(91):452–455, 1965. ISSN 00255718, 10886842. URL http://www.jstor.org/stable/2003676. 9

[5] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. 9

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. 1, 23

[7] Han Cai, Chuang Gan, and Song Han. Efficientvit: Enhanced linear attention for high-resolution low-computation visual recognition. *ArXiv*, abs/2205.14756, 2022. 3

[8] Ting-Yun Chang and Robin Jia. Data curation alone can stabilize in-context learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8123–8144, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.452. URL https://aclanthology.org/2023.acl-long.452. 23

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. 1

[10] Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability, 2023. 24

[11] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta-optimizers. *ArXiv*, abs/2212.10559, 2023. 1, 24

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423. 1

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy. 1

[14] Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *ArXiv*, abs/2208.01066, 2022. 1, 2, 4, 23

[15] Chi Han, Ziqi Wang, Han Zhao, and Heng Ji. In-context learning of large language models explained as kernel regression, 2023. 23

[16] Michael Hassid, Hao Peng, Daniel Rotem, Jungo Kasai, Ivan Montero, Noah A. Smith, and Roy Schwartz. How much does attention actually attend? questioning the importance of attention in pretrained transformers, 2022. 24

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9 (8):1735–1780, 11 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735. 10

[18] Yingcong Li, Muhammed Emrullah Ildiz, Dimitris Papailiopoulos, and Samet Oymak. Transformers as algorithms: Generalization and stability in in-context learning. In *International Conference on Machine Learning*, 2023. 23, 24

[19] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.deelio-1.10. URL https://aclanthology.org/2022.deelio-1.10. 23

[20] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8086–8098, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.556. URL https://aclanthology.org/2022.acl-long.556. 23

[21] Sewon Min, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Noisy channel language model prompting for few-shot text classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5316–5330, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.365. URL https://aclanthology.org/2022.acl-long.365. 23

[22] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. MetaICL: Learning to learn in context. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2791–2809, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.201. URL https://aclanthology.org/2022.naacl-main.201. 23

[23] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.759. URL https://aclanthology.org/2022.emnlp-main.759. 23

[24] E.H Moore. On the reciprocal of the general algebraic matrix. *Bulletin of American Mathematical Society*, 26:394–395, 1920. 9

[25] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability, 2023. 24

[26] Tai Nguyen and Eric Wong. In-context example selection with influences. *arXiv preprint arXiv:2302.11042*, 2023. 23

[27] OpenAI. Gpt-4 technical report, 2023. URL http://arxiv.org/abs/2303.08774v3. 1

[28] Victor Y. Pan and Robert S. Schreiber. An improved newton iteration for the generalized inverse of a matrix, with applications. *SIAM J. Sci. Comput.*, 12:1109–1130, 1991. 9

[29] Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets, 2022. 24

[30] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 2

[31] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher, 2022. 1

[32] Allan Raventós, Mansheej Paul, Feng Chen, and Surya Ganguli. Pretraining task diversity and the emergence of non-bayesian in-context learning for regression, 2023. 23

[33] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.191. URL https://aclanthology.org/2022.naacl-main.191. 23

[34] Günther Schulz. Iterative berechung der reziproken matrix. *Zeitschrift für Angewandte Mathematik und Mechanik (Journal of Applied Mathematics and Mechanics)*, 13:57–59, 1933. 9

[35] Vatsal Sharan, Aaron Sidford, and Gregory Valiant. Memory-sample tradeoffs for linear regression with small error. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 890–901, 2019. 14

[36] Kai Shen, Junliang Guo, Xu Tan, Siliang Tang, Rui Wang, and Jiang Bian. A study on relu and softmax in transformer, 2023. 3

[37] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022. 1

[38] Torsten Soderstrom and G. W. Stewart. On the numerical properties of an iterative method for computing the moore- penrose generalized inverse. *SIAM Journal on Numerical Analysis*, 11 (1):61–74, 1974. ISSN 00361429. URL http://www.jstor.org/stable/2156431. 9

[39] Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Selective annotation makes language models better few-shot learners. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=qY1hlv7gwg. 23

[40] Davoud Ataee Tarzanagh, Yingcong Li, Christos Thrampoulidis, and Samet Oymak. Transformers as support vector machines. *ArXiv*, abs/2308.16898, 2023. 1, 23, 24

[41] Davoud Ataee Tarzanagh, Yingcong Li, Xuechen Zhang, and Samet Oymak. Max-margin token selection in attention mechanism, 2023. 1, 23

[42] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022. 1

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 1, 3

[44] Johannes von Oswald, Eyvind Niklasson, E. Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, 2022. 1, 11, 23

[45] Johannes von Oswald, Eyvind Niklasson, Maximilian Schlegel, Seijin Kobayashi, Nicolas Zucchet, Nino Scherrer, Nolan Miller, Mark Sandler, Blaise Agüera y Arcas, Max Vladymyrov, Razvan Pascanu, and Joao Sacramento. Uncovering mesa-optimization algorithms in transformers. *ArXiv*, abs/2309.05858, 2023. 1, 23

[46] Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: a circuit for indirect object identification in gpt-2 small, 2022. 24

[47] Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, and Tengyu Ma. Larger language models do in-context learning differently, 2023. 23

[48] Mitchell Wortsman, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Replacing softmax with relu in vision transformers, 2023. 3

[49] Kang Min Yoo, Junyeob Kim, Hyuhng Joon Kim, Hyunsoo Cho, Hwiyeol Jo, Sang-Woo Lee, Sang-goo Lee, and Taeuk Kim. Ground-truth labels matter: A deeper look into input-label demonstrations. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2422–2437, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.155. URL https://aclanthology.org/2022.emnlp-main.155. 23

[50] Ruiqi Zhang, Spencer Frei, and Peter L. Bartlett. Trained transformers learn linear models in-context. *ArXiv*, abs/2306.09927, 2023. 23

[51] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, pages 12697–12706. PMLR, 2021. 23

Appendix

A Problem Setup and Known Algorithms

A.1 Standard Methods for Solving Linear Regression

Here we discuss various known algorithms we compare Transformers with.

For any time step t, let $\boldsymbol{X}^{(t)} = [\boldsymbol{x}_1 \quad \cdots \quad \boldsymbol{x}_t]^\top$ be the data matrix and $\boldsymbol{y}^{(t)} = [y_1 \quad \cdots \quad y_t]^\top$ be the labels for all the datapoints seen so far. Note that since t can be smaller than the data dimension d, $\boldsymbol{X}^{(t)}$ can be singular. We now consider various algorithms for making predictions for $\boldsymbol{x}_{t+1}$ based on $\boldsymbol{X}^{(t)}$ and $\boldsymbol{y}^{(t)}$. When it is clear from context, we will drop the superscript and refer to $\boldsymbol{X}^{(t)}$ and $\boldsymbol{y}^{(t)}$ as $\boldsymbol{X}$ and $\boldsymbol{y}$, where $\boldsymbol{X}$ and $\boldsymbol{y}$ correspond to all the datapoints seen so far.

Ordinary Least Squares. This method finds the minimum-norm solution to the objective:

$$\mathcal{L}(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2n}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2. \tag{6}$$

The Ordinary Least Squares (OLS) solution has a closed form given by the Normal Equations:

$$\hat{\boldsymbol{w}}^{\mathrm{OLS}} = (\boldsymbol{X}^\top \boldsymbol{X})^\dagger \boldsymbol{X}^\top \boldsymbol{y} \tag{7}$$

where $\boldsymbol{S} := \boldsymbol{X}^\top \boldsymbol{X}$ and $\boldsymbol{S}^\dagger$ is the pseudo-inverse [24] of $\boldsymbol{S}$.

Gradient Descent. Gradient descent (GD) finds the weight vector $\hat{\boldsymbol{w}}^{\mathrm{GD}}$ by with randomly initialized $\hat{\boldsymbol{w}}_0^{\mathrm{GD}}$ and using the iterative update rule:

$$\hat{\boldsymbol{w}}_{k+1}^{\mathrm{GD}} = \hat{\boldsymbol{w}}_k^{\mathrm{GD}} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\hat{\boldsymbol{w}}_k^{\mathrm{GD}} \mid \boldsymbol{X}, \boldsymbol{y}). \tag{8}$$

It is known that Gradient Descent requires $\mathcal{O}\left(\kappa(\boldsymbol{S}) \log(1/\epsilon)\right)$ steps to converge to an ϵ error where $\kappa(\boldsymbol{S}) = \frac{\lambda_{\max}(\boldsymbol{S})}{\lambda_{\min}(\boldsymbol{S})}$ is the *condition number*. Thus, when $\kappa(\boldsymbol{S})$ is large, Gradient Descent converges slowly [5].

Online Gradient Descent. While GD computes the gradient with respect to the full data matrix $\boldsymbol{X}$ at each iteration, Online Gradient Descent (OGD) is an online algorithm that only computes gradients on the newly received data point $\{\boldsymbol{x}_k, y_k\}$ at step k:

$$\hat{\boldsymbol{w}}_{k+1}^{\mathrm{OGD}} = \hat{\boldsymbol{w}}_k^{\mathrm{OGD}} - \eta_k \nabla_{\boldsymbol{w}} \mathcal{L}(\hat{\boldsymbol{w}}_k^{\mathrm{OGD}} \mid \boldsymbol{x}_k, y_k). \tag{9}$$

Picking $\eta_k = \frac{1}{\|\boldsymbol{x}_k\|_2^2}$ ensures that the new weight vector $\hat{\boldsymbol{w}}_{k+1}^{\mathrm{OGD}}$ makes zero error on $\{\boldsymbol{x}_k, y_k\}$.

Iterative Newton's Method. This method finds the weight vector $\hat{\boldsymbol{w}}^{\mathrm{Newton}}$ by iteratively apply Newton's method to finding the pseudo inverse of $\boldsymbol{S} = \boldsymbol{X}^\top \boldsymbol{X}$ [34, 4].

$$\begin{aligned} &\boldsymbol{M}_0 = \alpha \boldsymbol{S}, \text{ where } \alpha = \frac{2}{\|\boldsymbol{S}\boldsymbol{S}^\top\|_2}, && \hat{\boldsymbol{w}}_0^{\mathrm{Newton}} = \boldsymbol{M}_0 \boldsymbol{X}^\top \boldsymbol{y}, \\ &\boldsymbol{M}_{k+1} = 2\boldsymbol{M}_k - \boldsymbol{M}_k \boldsymbol{S} \boldsymbol{M}_k, && \hat{\boldsymbol{w}}_{k+1}^{\mathrm{Newton}} = \boldsymbol{M}_{k+1} \boldsymbol{X}^\top \boldsymbol{y}. \end{aligned} \tag{10}$$

This computes an approximation of the psuedo inverse using the moments of $\boldsymbol{S}$. In contrast to GD, the Iterative Newton's method only requires $\mathcal{O}(\max\{\log \kappa(\boldsymbol{S}), \log\log(1/\epsilon)\})$ steps to converge to an ϵ error [38, 28].

A.2 LSTM

While our primary goal is to analyze Transformers, we also consider the LSTM architecture [17] to understand whether Transformers learn different algorithms than other neural sequence models trained to do linear regression. In particular, we train a unidirectional L-layer LSTM, which generates a sequence of hidden states $\boldsymbol{H}^{(\ell)}$ for each layer ℓ, similarly to an L-layer Transformer. As with Transformers, we add a readout layer that predicts the $\hat{y}_{t+1}^{\mathrm{LSTM}}$ from the final hidden state at the final layer, $\boldsymbol{H}_{:,2t+1}^{(L)}$.

A.3 Measuring Algorithmic Similarity

A.3.1 Metrics

We propose two metrics to measure the similarity between linear regression algorithms.

Similarity of Errors. For a linear regression algorithm $\mathcal{A}$, let $\mathcal{A}(\boldsymbol{x}_{t+1} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t)$ denote its prediction on the $(t+1)$-th in-context example $\boldsymbol{x}_{t+1}$ after observing the first t examples (see Figure 1). We write $\mathcal{A}(\boldsymbol{x}_{t+1}) := \mathcal{A}(\boldsymbol{x}_{t+1} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t)$ for brevity. The errors (i.e., residuals) on the data sequence are:[1]

$$\mathcal{E}(\mathcal{A} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^{n+1}) = \Big[\mathcal{A}(\boldsymbol{x}_2) - y_2, \cdots, \mathcal{A}(\boldsymbol{x}_{n+1}) - y_{n+1}\Big]^\top \in \mathbb{R}^n. \tag{11}$$

For any two algorithms $\mathcal{A}_a$ and $\mathcal{A}_b$, their similarity of errors, corresponding to the metric $\mathcal{C}(\cdot, \cdot)$, is

$$\mathrm{SimE}(\mathcal{A}_a, \mathcal{A}_b) = \mathop{\mathbb{E}}_{\{\boldsymbol{x}_i, y_i\}_{i=1}^{n+1} \sim P_{\mathcal{D}}} \mathcal{C}\Big(\mathcal{E}(\mathcal{A}_a \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^{n+1}), \mathcal{E}(\mathcal{A}_b \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^{n+1})\Big) \tag{12}$$

where we use the cosine similarity as our correlation metric $\mathcal{C}(\boldsymbol{u}, \boldsymbol{v}) = \dfrac{\langle \boldsymbol{u}, \boldsymbol{v} \rangle}{\|\boldsymbol{u}\|_2 \|\boldsymbol{v}\|_2}$. Here n is the total number of in-context examples and $P_{\mathcal{D}}$ is the data generation process discussed previously.

Similarity of Induced Weights. All standard algorithms for linear regression estimate a weight vector $\hat{\boldsymbol{w}}$. While neural ICL models like Transformers do not explicitly learn such a weight vector, similar to [2], we can *induce* an implicit weight vector $\tilde{\boldsymbol{w}}$ learned by any algorithm $\mathcal{A}$ by fitting a weight vector to its predictions. To do this, for any fixed sequence of t in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^t$, we sample $T \gg d$ query examples $\tilde{\boldsymbol{x}}_k \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$, where $k \in [T]$. For this fixed sequence of in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^t$, we create T in-context prediction tasks and use the algorithm $\mathcal{A}$ to make predictions $\mathcal{A}(\tilde{\boldsymbol{x}}_k \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t)$. We define the induced data matrix and labels as

$$\tilde{\boldsymbol{X}} = \begin{bmatrix} \tilde{\boldsymbol{x}}_1^\top \\ \vdots \\ \tilde{\boldsymbol{x}}_T^\top \end{bmatrix} \qquad \tilde{\boldsymbol{Y}} = \begin{bmatrix} \mathcal{A}(\tilde{\boldsymbol{x}}_1 \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t) \\ \vdots \\ \mathcal{A}(\tilde{\boldsymbol{x}}_T \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t) \end{bmatrix}. \tag{13}$$

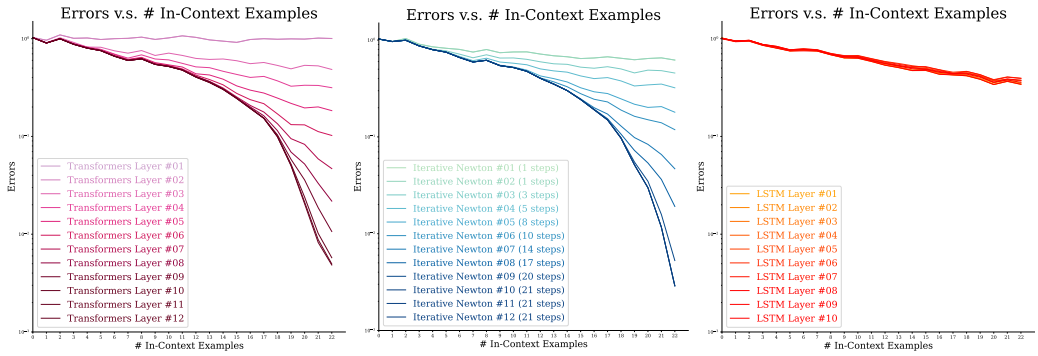
Then, the induced weight vector for $\mathcal{A}$ and these t in-context examples is:

$$\tilde{\boldsymbol{w}}_t(\mathcal{A}) := \tilde{\boldsymbol{w}}_t(\mathcal{A} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t) = (\tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{X}})^{-1} \tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{Y}}. \tag{14}$$

We measure the similarity between two algorithms $\mathcal{A}_a$ and $\mathcal{A}_b$ by measuring the similarity of induced weight vectors $\tilde{\boldsymbol{w}}_t(\mathcal{A}_a)$ and $\tilde{\boldsymbol{w}}_t(\mathcal{A}_b)$. We define the similarity of induced weights between two algorithms as

$$\mathrm{SimW}(\mathcal{A}_a, \mathcal{A}_b) = \mathop{\mathbb{E}}_{\{\boldsymbol{x}_i, y_i\}_{i=1}^{n} \sim P_{\mathcal{D}}} \frac{1}{n} \sum_{t=1}^{n} \mathcal{C}\Big(\tilde{\boldsymbol{w}}_t(\mathcal{A}_a \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t), \tilde{\boldsymbol{w}}_t(\mathcal{A}_b \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t)\Big).$$

[1] the indices start from 2 to $n+1$ because we evaluate all cases where t can choose from $1, \cdots, n$.



(a) Transformers (b) Iterative Newton's Method (c) LSTM

Figure 3: **Progression of Algorithms.** (a) Transformer's performance improves over the layer index ℓ. (b) Iterative Newton's performance improves over the number of iterations k, in a way that closely resembles the Transformer. We plot the best-matching k to Transformer's ℓ following Definition 4. (c) In contrast, LSTM's performance does not improve from layer to layer.

A.3.2 Best Matching Hyper-parameters Between Algorithms

Each algorithm we consider has its own hyper-parameter(s), for example the number of iterations for Iterative Newton and Gradient Descent, and the number of layers for Transformers (see Appendix B.1). When comparing two algorithms, given a choice of hyper-parameters for the first algorithm, we compare with the hyper-parameters for the second algorithm that maximize algorithmic similarity. In other words, we measure whether there exists hyperparameters for the second algorithm that make the two algorithms are similar.

Definition 4 (Hyper-Parameter Matching)**.** *Let $\mathcal{M}$ be the metric for evaluating similarities between two algorithms $\mathcal{A}_a$ and $\mathcal{A}_b$, which have hyper-parameters $p_a \in \mathcal{P}_a$ and $p_b \in \mathcal{P}_b$, respectively. For a given choice of p_a, We define the best-matching hyper-parameters of algorithm $\mathcal{A}_b$ for $\mathcal{A}_a$ as:*

$$p_b^{\mathcal{M}}(p_a) := \arg\min_{p_b \in \mathcal{P}_b} \mathcal{M}(\mathcal{A}_a(\cdot \mid p_a), \mathcal{A}_b(\cdot \mid p_b)). \tag{15}$$

The matching processes can be visualized as heatmaps as shown in Figure 2, where best-matching hyperparameters are highlighted. We will discuss these results further in §4.

B Additional Experimental Results

B.1 Transformers improve progressively over layers

Many known algorithms for linear regression, including GD, OGD, and Iterative Newton, are *iterative*: their performance progressively improves as they perform more iterations, eventually converging to a final solution. How could a Transformer implement such an iterative algorithm? von Oswald et al. [44] propose that deeper *layers* of the Transformer may correspond to more iterations of an iterative method; in particular, they show that there exist Transformer parameters such that each attention layer performs one step of Gradient Descent.

Following this intuition, we first investigate whether the predictions of a trained Transformer improve as the layer index ℓ increases. For each layer of hidden states $\boldsymbol{H}^{(\ell)}$ (defined in definition 2), we re-train the `ReadOut` layer to predict y_t for each t; the new predictions are given by $\mathrm{ReadOut}^{(\ell)}\left[\boldsymbol{H}^{(\ell)}\right]$. Thus for each input prompt, there are L Transformer predictions parameterized by layer index ℓ. All parameters besides the `Readout` layer parameters are kept frozen.

As shown in Figure 3a, as we increase the layer index ℓ, the prediction performance improves progressively. Hence, Transformers progressively improve their predictions over layers ℓ, similar to how iterative algorithms improve over steps.

B.2 Additional Results on Isotropic Data

B.2.1 Heatmaps

We present heatmaps with all values of similarities.

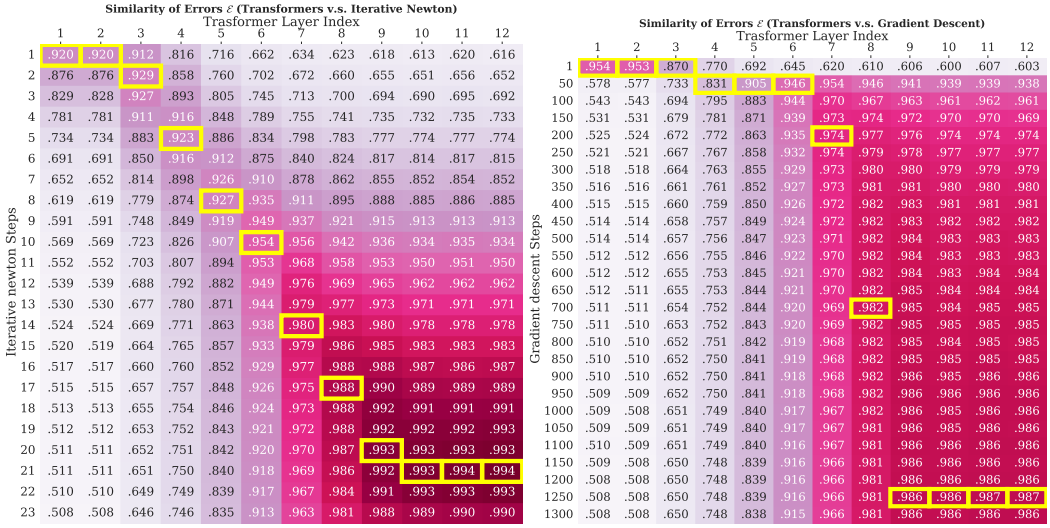


Figure 4: **Similarity of Errors.** The best matching hyper-parameters are highlighted in yellow.

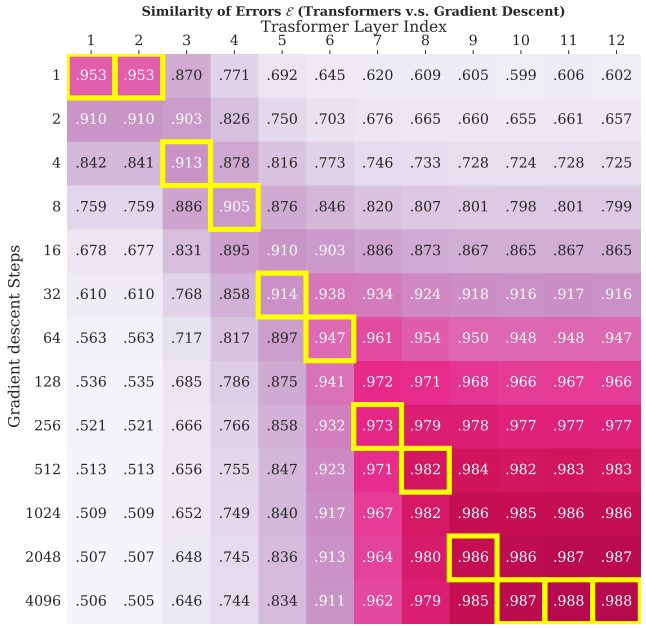


Figure 5: **Similarity of Errors of Gradient Descent in Log Scale.** The best matching hyper-parameters are highlighted in yellow. Putting the number of steps of Gradient Descent in log scale further verifies the claim that Transformer's rate of covergence is exponentially faster than that of Gradient Descent.

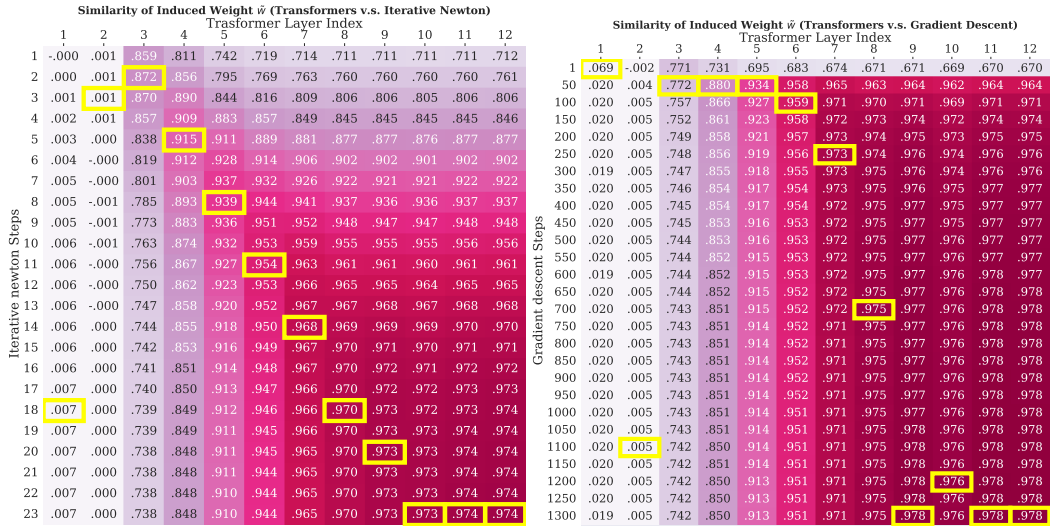


Figure 6: **Similarity of Induced Weight Vectors.** The best matching hyper-parameters are highlighted in yellow.

B.2.2 Additional Results on Comparison over Transformer Layers

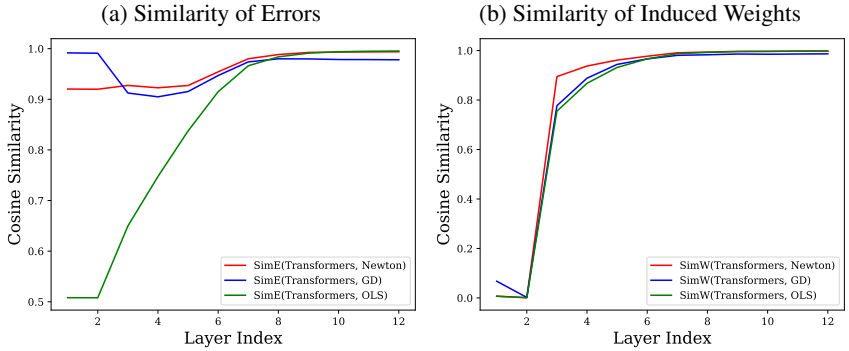


Figure 7: Similarities between Transformer and candidate algorithms. Transformers resemble *Iterative Newton's Method* the most.

B.2.3 Additional Results on Similarity of Induced Weights

We present more details line plots for how the similarity of weights changes as the models see more in-context observations $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$, i.e., as n increases. We fix the number of Transformers layers ℓ and compare with other algorithms with their best-match hyperparamters to ℓ in Figure 8.

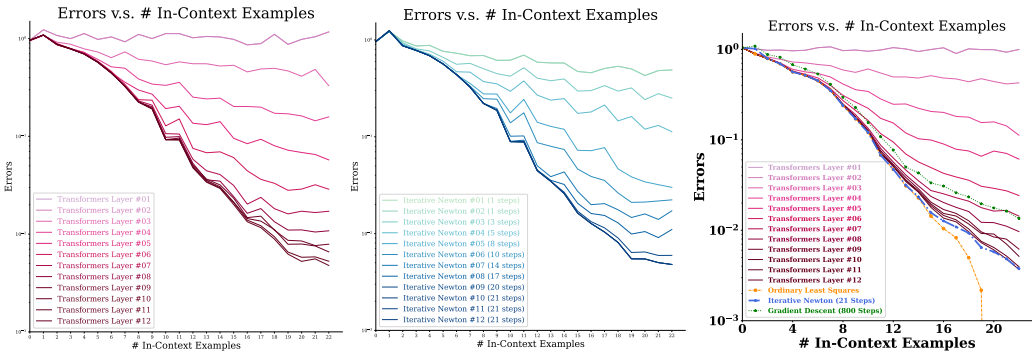
B.3 Experiments on Ill-Conditioned Problems

We repeat the same experiments with data $\boldsymbol{x}_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$ sampled from an ill-condition covariance matrix $\boldsymbol{\Sigma}$ with condition number $\kappa(\boldsymbol{\Sigma}) = 100$, and eigenbasis chosen uniformly at random. The first $d/2$ eigenvalues of $\boldsymbol{\Sigma}$ are 100, the last $d/2$ are 1.

As shown in Figure 9c, the Transformer model's performance still closely matches Iterative Newton's Method with 21 iterations, same as when $\boldsymbol{\Sigma} = \boldsymbol{I}$ (see layer 10-12 in Figure 2). The convergence of higher-order methods has a mild logarithmic dependence on the condition number since they correct for the curvature. On the other hand, Gradient Descent's convergence is affected polynomially by conditioning. As $\kappa(\boldsymbol{\Sigma})$ increase from 1 to 100, the number steps required for GD's convergence

increases significantly (see Fig. 9c where GD requires 800 steps to converge), making it impossible for a 12-layer Transformers to implement these many gradient updates. We also note that preconditioning the data by $(\boldsymbol{X}^\top \boldsymbol{X})^\dagger$ can make the data well-conditioned, but since the eigenbasis is chosen uniformly at random, with high probability there is no sparse pre-conditioner or any fixed pre-conditioner which works across the data distribution. Computing $(\boldsymbol{X}^\top \boldsymbol{X})^\dagger$ appears to be as hard as computing the OLS solution (Eq. 6)—in fact Sharan et al. [35] conjecture that first-order methods such as gradient descent and its variants cannot avoid polynomial dependencies in condition number in the ill-conditioned case.

These experiments on ill-conditioned data further strengthen our hypothesis that Transformers are learning to perform higher-order optimization methods in-context, not Gradient Descent.



(a) Transformers (b) Iterative Newton's Method (c) Comparisons

Figure 9: **Progression of Algorithms on Ill-Conditioned Data.** Transformer's performance still improves over the layer index ℓ; Iterative Newton's Method's performance improves over the number of iterations t and we plot the best-matching t to Transformer's ℓ following definition 4.

We also present the heatmaps to find the best-matching hyper-parameters and conclude that Transformers are similar to Newton's method than GD in ill-conditioned data.

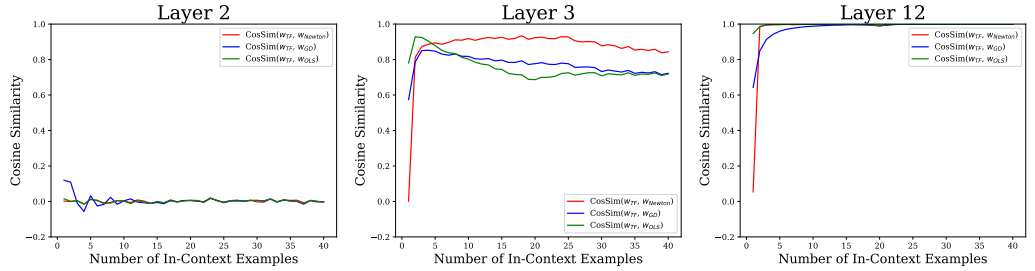


Figure 8: *Similarity of induced weights* over varying number of in-context examples, on three layer indices of Transformers, indexed as 2, 3 and 12. We find that initially at layer 2, the Transformers model hasn't learned so it has zero similarity to all candidate algorithms. As we progress to the next layer number 3, we find that Transformers start to learn, and when provided few examples, Transformers are more similar to OLS but soon become most similar to the Iterative Newton's Method. Layer 12 shows that Transformers in the later layers converge to the OLS solution when provided more than 1 example. We also find there is a dip around $n = d$ for similarity between Transformers and OLS but not for Transformers and Newton, and this is probably because OLS has a more prominent double-descent phenomenon than Transformers and Newton.

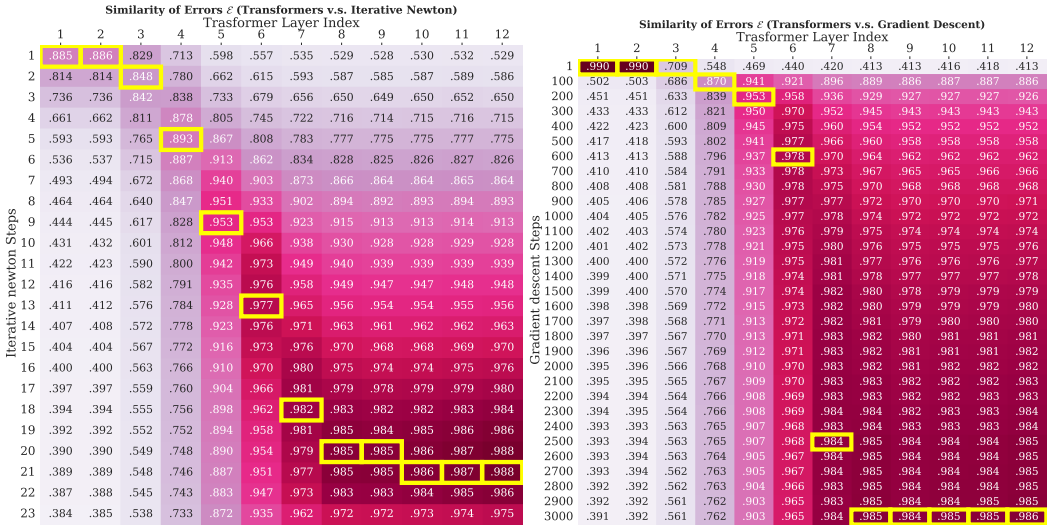


Figure 10: **Similarity of Errors on Ill-Conditioned Data.** The best matching hyper-parameters are highlighted in yellow.

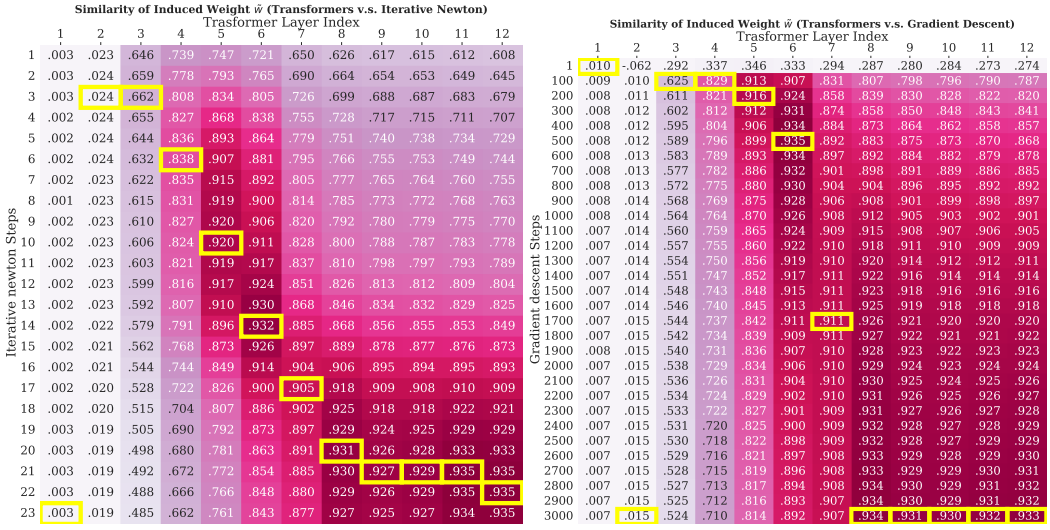


Figure 11: **Similarity of Induced Weights on Ill-Conditioned Data.** The best matching hyper-parameters are highlighted in yellow.

B.4 Experiments on alternative configurations

In this section, we present experimental results from an alternative model configurations than the main text. We show in the main text that Transformers learn higher-order optimization methods in-context where the experiments are using a GPT-2 model with 12 layers and 8 heads per layer. In this section, we present experiments with a GPT-2 model with 12 layers but only 1 head per layer.

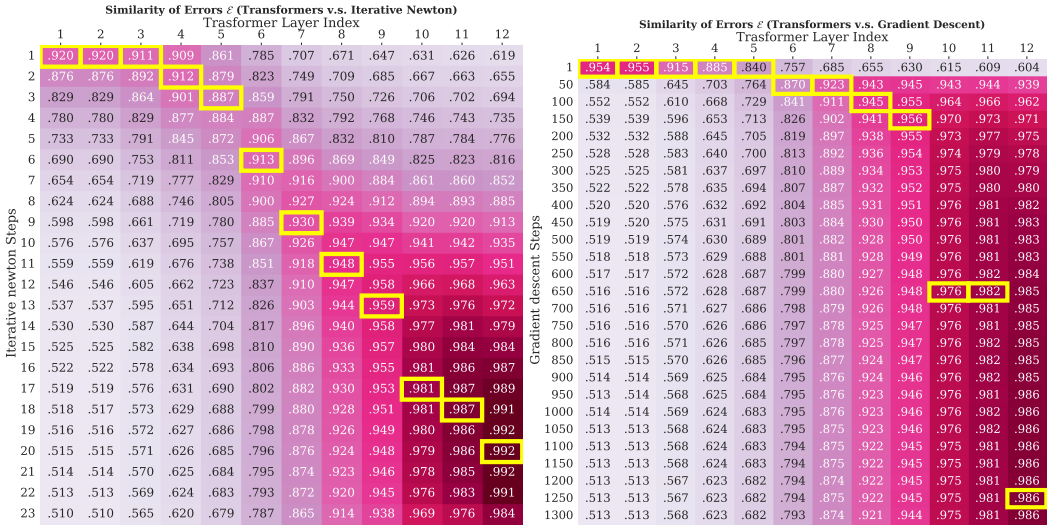


Figure 12: **Similarity of Errors on an alternative Transformers Configuration.** The best matching hyper-parameters are highlighted in yellow.

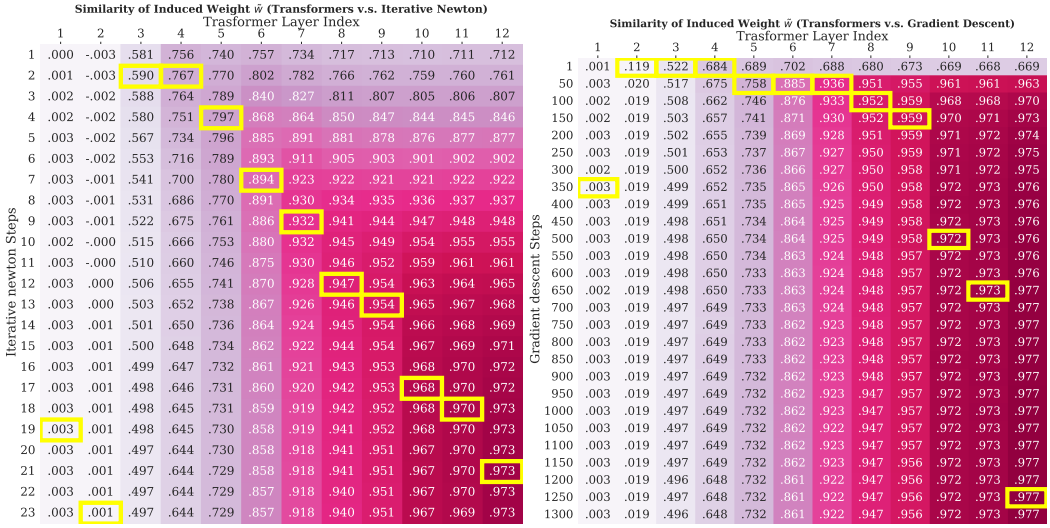


Figure 13: **Similarity of Induced Weights on an alternative Transformers Configuration.** The best matching hyper-parameters are highlighted in yellow.

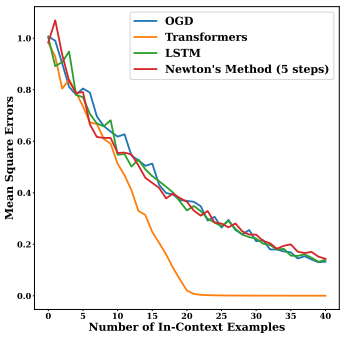
We conclude that our experimental results are not restricted to a specific model configurations, smaller models such as GPT-2 with 12 layers and 1 head each layer also suffice in implementing the Newton's method, and more similar than gradient descents.

B.5 Definitions for Evaluating Forgetting

We measure the phenomenon of model forgetting by reusing an in-context example within $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$ as the test example $\boldsymbol{x}_{\text{test}}$. In experiments of Figure 14, we fix $n = 20$ and reuse $\boldsymbol{x}_{\text{test}} = \boldsymbol{x}_i$. We denote the "Time Stamp Gap" as the distance the reused example index i from the current time stamp $n = 20$. We measure the forgetting of index i as

$$\text{Forgetting}(\mathcal{A}, i) = \mathop{\mathbb{E}}_{\{\boldsymbol{x}_i, y_i\}_{i=1}^n \sim P_{\mathcal{D}}} \text{MSE}\Big(\mathcal{A}(\boldsymbol{x}_i \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^n), y_i\Big) \tag{16}$$

Note: the further away i is from n, the more possible algorithm $\mathcal{A}$ forgets.



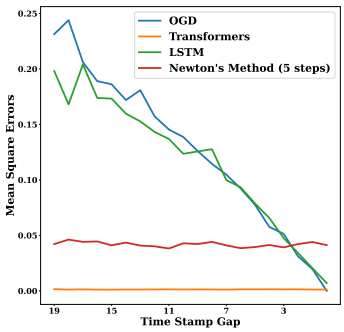


Table 1: **Similarity of errors between algorithms.** Transformers are more similar to full-observation methods such as Newton and GD; and LSTMs are more similar to online methods such as OGD.

	Transformers	LSTM
Newton	**0.991**	0.920
GD	**0.862**	0.808
OGD	0.667	**0.831**

Figure 14: In the left figure, we measure model predictions with normalized MSE. Though LSTM is seemingly most similar to Newton's Method with only 5 steps, neither algorithm converges yet. OGD also has a similar trend as LSTM. In the center figure, we measure model's forgetting phenomenon (see Appendix B.5 for explanations), and find both Transformers and not-converged Newton have better memorization than LSTM and OGD. In the right table, we find that Transformers are more similar to Newton and GD than LSTMs do while LSTM is significantly more similar to OGD.

B.6 LSTM is more similar to OGD than Transformers

As discussed in Section 2, LSTM is an alternative auto-regressive model widely used before the introduction of Transformers. Thus, a natural research question is: *If Transformers can learn in-context, can LSTMs do so as well? If so, do they learn the same algorithms?* To answer this question, we train a 10-layer LSTM model, with 5.3M parameters, in an identical manner to the Transformers (with 9.5M parameters) studied in the previous sections.[2]

Figure 14 plots the mean squared error of Transformers, LSTMs, and other standard methods as a function of the number of in-context (i.e., training) examples provided. While LSTMs can also learn linear regression in-context, they have much higher mean-squared error than Transformers. Their error rate is similar to Iterative Newton's Method after only 5 iterations, a point where it is far from converging to the OLS solution.

LSTMs' inferior performance to Transformers can be explained by the inability of LSTMs to use deeper layers to improve their predictions. Figure 3 shows that LSTM performance does not improve across layers—a readout head fine-tuned for the first layer makes equally good predictions as the full 10-layer model. Thus, LSTMs seem poorly equipped to fully implement iterative algorithms.

Finally, we show that LSTMs behave more like an online learning algorithm than Transformers. In particular, its predictions are biased towards getting more recent training examples correct, as opposed to earlier examples, as shown in Figure 14. This property makes LSTMs similar to online Gradient Descent. In contrast, five steps of Newton's method has the same error on average for recent and early examples, showing that the LSTM implements a very different algorithm from a few iterations of Newton. Similarly, Table 1 shows that LSTMs are more similar to OGD than Transformers are, whereas Transformers are more similar to Newton and GD than LSTMs. We hypothesize that since LSTMs have limited memory, they must learn in a roughly online fashion; in contrast, Transformers' attention heads can access the entire sequence of past examples, enabling it to learn more complex algorithms.

[2]While the LSTM has fewer parameters than the Transformer, we found in preliminary experiments that increasing the hidden dimension or number of layers in the LSTM would not substantively change our results.

C Detailed Proofs for Section 3

In this section, we work on full attention layers with normalized ReLU activation $\sigma(\cdot) = \frac{1}{n}\mathrm{ReLU}(\cdot)$ given n examples.

Definition 5. *A full attention layer with M heads and ReLU activation is also denoted as* Attn *on any input sequence $\boldsymbol{H} = [\boldsymbol{h}_1, \cdots, \boldsymbol{h}_N] \in \mathbb{R}^{D\times N}$, where D is the dimension of hidden states and N is the sequence length. In the vector form,*

$$\tilde{\boldsymbol{h}}_t = [\mathrm{Attn}(\boldsymbol{H})]_t = \boldsymbol{h}_t + \frac{1}{n}\sum_{m=1}^{M}\sum_{j=1}^{n}\mathrm{ReLU}\left(\langle \boldsymbol{Q}_m\boldsymbol{h}_t, \boldsymbol{K}_m\boldsymbol{h}_j\rangle\right)\cdot \boldsymbol{V}_m\boldsymbol{h}_j \tag{17}$$

Remark 1. *This is slightly different from the **causal** attention layer (see definition 1) in that at each time stamp t, the attention layer in definition 5 has full information of all hidden states $j \in [n]$, unlike causal attention layer which requires $j \in [t]$.*

C.1 Helper Results

We begin by constructing a useful component for our proof, and state some existing constructions from Akyürek et al. [2].

Lemma 1. *Given hidden states $\{\boldsymbol{h}_1, \cdots, \boldsymbol{h}_n\}$, there exists query, key and value matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ respectively such that one attention layer can compute $\sum_{j=1}^{n}\boldsymbol{h}_j$.*

Proof. We can pad each hidden state by 1 and 0's such that $\boldsymbol{h}'_t \leftarrow \begin{bmatrix}\boldsymbol{h}_t\\ 1\\ \boldsymbol{0}_d\end{bmatrix} \in \mathbb{R}^{2d+1}$. We construct two heads where $\boldsymbol{Q}_1 = \boldsymbol{K}_1 = \boldsymbol{Q}_2 = \begin{bmatrix}\boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times 1} & \boldsymbol{O}_{d\times d}\\ \boldsymbol{O}_{1\times d} & 1 & \boldsymbol{O}_{1\times d}\\ \boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times 1} & \boldsymbol{O}_{d\times d}\end{bmatrix}$ and $\boldsymbol{K}_2 = -\boldsymbol{K}_1$. Then $\begin{bmatrix}\boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times 1} & \boldsymbol{O}_{d\times d}\\ \boldsymbol{O}_{1\times d} & 1 & \boldsymbol{O}_{1\times d}\\ \boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times 1} & \boldsymbol{O}_{d\times d}\end{bmatrix}\boldsymbol{h}'_t = \begin{bmatrix}\boldsymbol{0}_d\\ 1\\ \boldsymbol{0}_d\end{bmatrix}$.

Let $\boldsymbol{V}_1 = \boldsymbol{V}_2 = \begin{bmatrix}\boldsymbol{O}_{(d+1)\times d} & \boldsymbol{O}_{(d+1)\times(d+1)}\\ n\boldsymbol{I}_{d\times d} & \boldsymbol{O}_{d\times(d+1)}\end{bmatrix}$ so that $\boldsymbol{V}_m\begin{bmatrix}\boldsymbol{h}_j\\ 1\\ \boldsymbol{0}_d\end{bmatrix} = \begin{bmatrix}\boldsymbol{0}_{d+1}\\ n\boldsymbol{h}_j\end{bmatrix}$.

We apply one attention layer to these 1-padded hidden states and we have

$$\begin{aligned}\tilde{\boldsymbol{h}}_t &= \boldsymbol{h}'_t + \frac{1}{n}\sum_{m=1}^{2}\sum_{j=1}^{n}\mathrm{ReLU}\left(\langle \boldsymbol{Q}_m\boldsymbol{h}'_t, \boldsymbol{K}_m\boldsymbol{h}'_j\rangle\right)\cdot \boldsymbol{V}_m\boldsymbol{h}'_j\\ &= \boldsymbol{h}'_t + \frac{1}{n}\sum_{j=1}^{n}\left[\mathrm{ReLU}(1) + \mathrm{ReLU}(-1)\right]\cdot\begin{bmatrix}\boldsymbol{0}_{d+1}\\ n\boldsymbol{h}_j\end{bmatrix}\\ &= \begin{bmatrix}\boldsymbol{h}_t\\ 1\\ \boldsymbol{0}_d\end{bmatrix} + \begin{bmatrix}\boldsymbol{0}_{d+1}\\ \sum_{j=1}^{n}\boldsymbol{h}_j\end{bmatrix} = \begin{bmatrix}\boldsymbol{h}_t\\ 1\\ \sum_{j=1}^{n}\boldsymbol{h}_j\end{bmatrix}\end{aligned} \tag{18}$$

□

Proposition 1 (2)**.** *Each of* `mov`, `aff`, `mul`, `div` *can be implemented by a single transformer layer. These four operations are mappings $\mathbb{R}^{D\times N} \to \mathbb{R}^{D\times N}$, expressed as follows,*

`mov`$(\boldsymbol{H}; s, t, i, j, i', j')$*: selects the entries of the s-th column of $\boldsymbol{H}$ between rows i and j, and copies them into the t-th column ($t \geq s$) of $\boldsymbol{H}$ between rows i' and j'.*

`mul`$(\boldsymbol{H}; a, b, c, (i, j), (i', j'), (i'', j''))$*: in each column $\boldsymbol{h}$ of $\boldsymbol{H}$, interprets the entries between i and j as an $a \times b$ matrix $\boldsymbol{A}_1$, and the entries between i' and j' as a $b \times c$ matrix $\boldsymbol{A}_2$, multiplies these matrices together, and stores the result between rows i'' and j'', yielding a matrix in which each column has the form $[\boldsymbol{h}_{:i''-1}, \boldsymbol{A}_1\boldsymbol{A}_2, \boldsymbol{h}_{j'':}]^\top$. This allows the layer to implement inner products.*

$\texttt{div}(\boldsymbol{H}; (i,j), i', (i'', j''))$*: in each column* $\boldsymbol{h}$ *of* $\boldsymbol{H}$*, divides the entries between* i *and* j *by the absolute value of the entry at* i'*, and stores the result between rows* i'' *and* j''*, yielding a matrix in which every column has the form* $[\boldsymbol{h}_{:i''-1}, \boldsymbol{h}_{i:j}/|\boldsymbol{h}_{i'}|, \boldsymbol{h}_{j'':}]^\top$*.*

$\texttt{aff}(\boldsymbol{H}; (i,j), (i',j'), (i'',j''), \boldsymbol{W}_1, \boldsymbol{W}_2, \mathbf{b})$*: in each column* $\boldsymbol{h}$ *of* $\boldsymbol{H}$*, applies an affine transformation to the entries between* i *and* j *and* i' *and* j'*, then stores the result between rows* i'' *and* j''*, yielding a matrix in which every column has the form* $[\boldsymbol{h}_{:i''-1}, \boldsymbol{W}_1\boldsymbol{h}_{i:j} + \boldsymbol{W}_2\boldsymbol{h}_{i':j'} + \mathbf{b}, \boldsymbol{h}_{j'':}]^\top$*. This allows the layer to implement subtraction by setting* $\boldsymbol{W}_1 = \boldsymbol{I}$ *and* $\boldsymbol{W}_2 = -\boldsymbol{I}$*.*

C.2 Proof of Theorem 1

Theorem 1. *There exist Transformer weights such that on any set of in-context examples* $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$ *and test point* $\boldsymbol{x}_{\text{test}}$*, the Transformer predicts on* $\boldsymbol{x}_{\text{test}}$ *using* $\boldsymbol{x}_{\text{test}}^\top \hat{\boldsymbol{w}}_k^{\text{Newton}}$*. Here* $\hat{\boldsymbol{w}}_k^{\text{Newton}}$ *are the Iterative Newton updates given by* $\hat{\boldsymbol{w}}_k^{\text{Newton}} = \boldsymbol{M}_k \boldsymbol{X}^\top \boldsymbol{y}$ *where* $\boldsymbol{M}_j$ *is updated as*

$$\boldsymbol{M}_j = 2\boldsymbol{M}_{j-1} - \boldsymbol{M}_{j-1}\boldsymbol{S}\boldsymbol{M}_{j-1}, 1 \le j \le k, \quad \boldsymbol{M}_0 = \alpha \boldsymbol{S},$$

for some $\alpha > 0$ *and* $\boldsymbol{S} = \boldsymbol{X}^\top \boldsymbol{X}$*. The number of layers of the transformer is* $\mathcal{O}(k)$ *and the dimensionality of the hidden layers is* $\mathcal{O}(d)$*.*

Proof. We break the proof into parts.

Transformers Implement Initialization $\boldsymbol{T}^{(0)} = \alpha \boldsymbol{S}$.
Given input sequence $\boldsymbol{H} := \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\}$, with $\boldsymbol{x}_i \in \mathbb{R}^d$, we first apply the $\texttt{mov}$ operations given by Proposition 1 (similar to [2], we show only non-zero rows when applying these operations):

$$\begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ & & \end{bmatrix} \xrightarrow{\text{mov}} \begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix} \tag{19}$$

We call each column after $\texttt{mov}$ as $\boldsymbol{h}_j$. With an full attention layer, one can two heads with query and value matrices of the form $\boldsymbol{Q}_1^\top \boldsymbol{K}_1 = -\boldsymbol{Q}_2^\top \boldsymbol{K}_2 = \begin{bmatrix} \boldsymbol{I}_{d\times d} & \boldsymbol{O}_{d\times d} \\ \boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times d} \end{bmatrix}$ such that for any $t \in [n]$, we have

$$\sum_{m=1}^{2} \text{ReLU}\left(\langle \boldsymbol{Q}_m \boldsymbol{h}_t, \boldsymbol{K}_m \boldsymbol{h}_j \rangle\right) = \text{ReLU}(\boldsymbol{x}_t^\top \boldsymbol{x}_j) + \text{ReLU}(-\boldsymbol{x}_t^\top \boldsymbol{x}_j) = \langle \boldsymbol{x}_t, \boldsymbol{x}_j \rangle \tag{20}$$

Let all value matrices $\boldsymbol{V}_m = n\alpha \begin{bmatrix} \boldsymbol{I}_{d\times d} & \boldsymbol{O}_{d\times d} \\ \boldsymbol{O}_{d\times d} & \boldsymbol{O}_{d\times d} \end{bmatrix}$ for some $\alpha \in \mathbb{R}$. Combining the skip connections, we have

$$\tilde{\boldsymbol{h}}_t = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} + \frac{1}{n}\sum_{j=1}^{n} \langle \boldsymbol{x}_t, \boldsymbol{x}_j \rangle \, n\alpha \begin{bmatrix} \boldsymbol{x}_j \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} + \begin{bmatrix} \alpha \left(\sum_{j=1}^n \boldsymbol{x}_j \boldsymbol{x}_j^\top\right) \boldsymbol{x}_t \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_t + \alpha \boldsymbol{S}\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \tag{21}$$

Now we can use the $\texttt{aff}$ operator to make subtractions and then

$$\begin{bmatrix} \boldsymbol{x}_t + \alpha \boldsymbol{S}\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \xrightarrow{\text{aff}} \begin{bmatrix} (\boldsymbol{x}_t + \alpha \boldsymbol{S}\boldsymbol{x}_t) - \boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} = \begin{bmatrix} \alpha \boldsymbol{S}\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \tag{22}$$

We call this transformed hidden states as $\boldsymbol{H}^{(0)}$ and denote $\boldsymbol{T}^{(0)} = \alpha \boldsymbol{S}$:

$$\boldsymbol{H}^{(0)} = \begin{bmatrix} \boldsymbol{h}_1^{(0)} & \cdots & \boldsymbol{h}_n^{(0)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{T}^{(0)}\boldsymbol{x}_1 & \cdots & \boldsymbol{T}^{(0)}\boldsymbol{x}_n \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix} \tag{23}$$

Notice that $\boldsymbol{S}$ is symmetric and thereafter $\boldsymbol{T}^{(0)}$ is also symmetric.

Transformers implement Newton Iteration.
Let the input prompt be the same as Equation (23),

$$\boldsymbol{H}^{(0)} = \begin{bmatrix} \boldsymbol{h}_1^{(0)} & \cdots & \boldsymbol{h}_n^{(0)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{T}^{(0)}\boldsymbol{x}_1 & \cdots & \boldsymbol{T}^{(0)}\boldsymbol{x}_n \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix} \tag{24}$$

We claim that the ℓ's hidden states can be of the similar form

$$\boldsymbol{H}^{(\ell)} = \begin{bmatrix} \boldsymbol{h}_1^{(\ell)} & \cdots & \boldsymbol{h}_n^{(\ell)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_1 & \cdots & \boldsymbol{T}^{(\ell)}\boldsymbol{x}_n \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix} \tag{25}$$

We prove by induction that assuming our claim is true for ℓ, we work on $\ell+1$:

Let $\boldsymbol{Q}_m = \tilde{\boldsymbol{Q}}_m \underbrace{\begin{bmatrix} \boldsymbol{O}_d & -\frac{n}{2}\boldsymbol{I}_d \\ \boldsymbol{O}_d & \boldsymbol{O}_d \end{bmatrix}}_{\boldsymbol{G}}, \boldsymbol{K}_m = \tilde{\boldsymbol{K}}_m \underbrace{\begin{bmatrix} \boldsymbol{I}_d & \boldsymbol{O}_d \\ \boldsymbol{O}_d & \boldsymbol{O}_d \end{bmatrix}}_{\boldsymbol{J}}$ where $\tilde{\boldsymbol{Q}}_1^\top \tilde{\boldsymbol{K}}_1 := \boldsymbol{I}$, $\tilde{\boldsymbol{Q}}_2^\top \tilde{\boldsymbol{K}}_2 := -\boldsymbol{I}$ and $\boldsymbol{V}_1 = \boldsymbol{V}_2 = \underbrace{\begin{bmatrix} \boldsymbol{I}_d & \boldsymbol{O}_d \\ \boldsymbol{O}_d & \boldsymbol{O}_d \end{bmatrix}}_{\boldsymbol{J}}$. A 2-head self-attention layer, with ReLU attentions, can be written has

$$\boldsymbol{h}_t^{(\ell+1)} = [\mathrm{Attn}(\boldsymbol{H}^{(\ell)})]_t = \boldsymbol{h}_t^{(\ell)} + \frac{1}{n}\sum_{m=1}^{2}\sum_{j=1}^{n} \mathrm{ReLU}\left(\left\langle \boldsymbol{Q}_m \boldsymbol{h}_t^{(\ell)}, \boldsymbol{K}_m \boldsymbol{h}_j^{(\ell)} \right\rangle\right) \cdot \boldsymbol{V}_m \boldsymbol{h}_j^{(\ell)} \tag{26}$$

where

$$\begin{aligned}
&\sum_{m=1}^{2} \mathrm{ReLU}\left(\left\langle \boldsymbol{Q}_m \boldsymbol{h}_t^{(\ell)}, \boldsymbol{K}_m \boldsymbol{h}_j^{(\ell)} \right\rangle\right) \cdot \boldsymbol{V}_m \boldsymbol{h}_j^{(\ell)} \\
&= \left[\mathrm{ReLU}\Big((\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})^\top \underbrace{\tilde{\boldsymbol{Q}}_1^\top \tilde{\boldsymbol{K}}_1}_{\boldsymbol{I}} (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})\Big) + \mathrm{ReLU}\Big((\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})^\top \underbrace{\tilde{\boldsymbol{Q}}_2^\top \tilde{\boldsymbol{K}}_2}_{-\boldsymbol{I}} (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})\Big)\right] \cdot (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)}) \\
&= \left[\mathrm{ReLU}((\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})^\top (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})) + \mathrm{ReLU}(-(\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})^\top (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)}))\right] \cdot (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)}) \\
&= (\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})^\top (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})(\boldsymbol{J}\boldsymbol{h}_j^{(\ell)}) \\
&= (\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})(\boldsymbol{J}\boldsymbol{h}_j^{(\ell)})^\top (\boldsymbol{G}\boldsymbol{h}_t^{(\ell)})
\end{aligned} \tag{27}$$

Plug in our assumptions that $\boldsymbol{h}_j^{(\ell)} = \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{x}_j \end{bmatrix}$, we have $\boldsymbol{J}\boldsymbol{h}_j^{(\ell)} = \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{0}_d \end{bmatrix}$ and $\boldsymbol{G}\boldsymbol{h}_t^{(\ell)} = \begin{bmatrix} -\frac{n}{2}\boldsymbol{x}_t \\ \boldsymbol{0}_d \end{bmatrix}$, we have

$$\begin{aligned}
\boldsymbol{h}_t^{(\ell+1)} &= \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} + \frac{1}{n}\sum_{j=1}^{n} \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{0}_d \end{bmatrix} \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{0}_d \end{bmatrix}^\top \begin{bmatrix} -\frac{n}{2}\boldsymbol{x}_t \\ \boldsymbol{0}_d \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_t - \frac{1}{2}\sum_{j=1}^{n} (\boldsymbol{T}^{(\ell)}\boldsymbol{x}_j)(\boldsymbol{T}^{(\ell)}\boldsymbol{x}_j)^\top \boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_t - \frac{1}{2}\boldsymbol{T}^{(\ell)} \left(\sum_{j=1}^{n} \boldsymbol{x}_j \boldsymbol{x}_j^\top\right) {\boldsymbol{T}^{(\ell)}}^\top \boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \\
&= \begin{bmatrix} \left(\boldsymbol{T}^{(\ell)} - \frac{1}{2}\boldsymbol{T}^{(\ell)}\boldsymbol{S}{\boldsymbol{T}^{(\ell)}}^\top\right)\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix}
\end{aligned} \tag{28}$$

Now we pass over an MLP layer with

$$\boldsymbol{h}_t^{(\ell+1)} \leftarrow \boldsymbol{h}_t^{(\ell+1)} + \begin{bmatrix} \boldsymbol{I}_d & \boldsymbol{O}_d \\ \boldsymbol{O}_d & \boldsymbol{O}_d \end{bmatrix} \boldsymbol{h}_t^{(\ell+1)} = \begin{bmatrix} \left(2\boldsymbol{T}^{(\ell)} - \boldsymbol{T}^{(\ell)}\boldsymbol{S}{\boldsymbol{T}^{(\ell)}}^\top\right)\boldsymbol{x}_t \\ \boldsymbol{x}_t \end{bmatrix} \tag{29}$$

Now we denote the iteration

$$\boldsymbol{T}^{(\ell+1)} = 2\boldsymbol{T}^{(\ell)} - \boldsymbol{T}^{(\ell)}\boldsymbol{S}{\boldsymbol{T}^{(\ell)}}^\top \tag{30}$$

We find that ${\boldsymbol{T}^{(\ell+1)}}^\top = \boldsymbol{T}^{(\ell+1)}$ since $\boldsymbol{T}^{(\ell)}$ and $\boldsymbol{S}$ are both symmetric. It reduces to

$$\boldsymbol{T}^{(\ell+1)} = 2\boldsymbol{T}^{(\ell)} - \boldsymbol{T}^{(\ell)}\boldsymbol{S}\boldsymbol{T}^{(\ell)} \tag{31}$$

This is exactly the same as the Newton iteration.

Transformers can implement $\hat{\boldsymbol{w}}_\ell^{\mathrm{TF}} = \boldsymbol{T}^{(\ell)}\boldsymbol{X}^\top\boldsymbol{y}$.
Going back to the empirical prompt format $\{\boldsymbol{x}_1, y_1, \cdots, \boldsymbol{x}_n, y_n\}$. We can let parameters be zero

for positions of y's and only rely on the skip connection up to layer ℓ, and the $\boldsymbol{H}^{(\ell)}$ is then $\begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j & \boldsymbol{0} \\ \boldsymbol{x}_j & \boldsymbol{0} \\ 0 & y_j \end{bmatrix}_{j=1}^{n}$. We again apply operations from Proposition 1:

$$\begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j & \boldsymbol{0} \\ \boldsymbol{x}_j & \boldsymbol{0} \\ 0 & y_j \end{bmatrix}_{j=1}^{n} \xrightarrow{\text{mov}} \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j & \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{x}_j & \boldsymbol{0} \\ 0 & y_j \end{bmatrix}_{j=1}^{n} \xrightarrow{\text{mul}} \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j & \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{x}_j & \boldsymbol{0} \\ 0 & y_j \\ \boldsymbol{0} & \boldsymbol{T}^{(\ell)}y_j\boldsymbol{x}_j \end{bmatrix}_{j=1}^{n} \tag{32}$$

Now we apply Lemma 1 over all even columns in Equation (32) and we have

$$\text{Output} = \sum_{j=1}^{n} \begin{bmatrix} \boldsymbol{T}^{(\ell)}\boldsymbol{x}_j \\ \boldsymbol{0} \\ y_j \\ \boldsymbol{T}^{(\ell)}y_j\boldsymbol{x}_j \end{bmatrix} = \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{T}^{(\ell)}\sum_{j=1}^{n} y_j\boldsymbol{x}_j \end{bmatrix} = \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{T}^{(\ell)}\boldsymbol{X}^\top\boldsymbol{y} \end{bmatrix} \tag{33}$$

where $\boldsymbol{\xi}$ denotes irrelevant quantities. Note that the resulting $\boldsymbol{T}^{(\ell)}\boldsymbol{X}^\top\boldsymbol{y}$ is also the same as Iterative Newton's predictor $\hat{\boldsymbol{w}}_k = \boldsymbol{M}_k\boldsymbol{X}^\top\boldsymbol{y}$ after k iterations. We denote $\hat{\boldsymbol{w}}_\ell^{\text{TF}} = \boldsymbol{T}^{(\ell)}\boldsymbol{X}^\top\boldsymbol{y}$.

Transformers can make predictions on $\boldsymbol{x}_{test}$ by $\langle \hat{\boldsymbol{w}}_\ell^{\text{TF}}, \boldsymbol{x}_{\text{test}} \rangle$.

Now we can make predictions on text query $\boldsymbol{x}_{\text{test}}$:

$$\begin{bmatrix} \boldsymbol{\xi} & \boldsymbol{x}_{\text{test}} \\ \hat{\boldsymbol{w}}_\ell^{\text{TF}} & \boldsymbol{x}_{\text{test}} \end{bmatrix} \xrightarrow{\text{mov}} \begin{bmatrix} \boldsymbol{\xi} & \boldsymbol{x}_{\text{test}} \\ \hat{\boldsymbol{w}}_\ell^{\text{TF}} & \boldsymbol{x}_{\text{test}} \\ \boldsymbol{0} & \hat{\boldsymbol{w}}_\ell^{\text{TF}} \end{bmatrix} \xrightarrow{\text{mul}} \begin{bmatrix} \boldsymbol{\xi} & \boldsymbol{x}_{\text{test}} \\ \hat{\boldsymbol{w}}_\ell^{\text{TF}} & \boldsymbol{x}_{\text{test}} \\ \boldsymbol{0} & \hat{\boldsymbol{w}}_\ell^{\text{TF}} \\ 0 & \langle \hat{\boldsymbol{w}}_\ell^{\text{TF}}, \boldsymbol{x}_{\text{test}} \rangle \end{bmatrix} \tag{34}$$

Finally, we can have an readout layer $\boldsymbol{\beta}_{\text{ReadOut}} = \{\boldsymbol{u}, v\}$ applied (see definition 3) with $\boldsymbol{u} = [\boldsymbol{0}_{3d} \quad 1]^\top$ and $v = 0$ to extract the prediction $\langle \hat{\boldsymbol{w}}_\ell^{\text{TF}}, \boldsymbol{x}_{\text{test}} \rangle$ at the last location, given by $\boldsymbol{x}_{\text{test}}$. This is exactly how Iterative Newton makes predictions.

To Perform k steps of Newton's iterations, Transformers need $\mathcal{O}(k)$ layers.

Let's count the layers:

- **Initialization**: `mov` needs $\mathcal{O}(1)$ layer; gathering $\alpha\boldsymbol{S}$ needs $\mathcal{O}(1)$ layer; and `aff` needs $\mathcal{O}(1)$ layer. In total, Transformers need $\mathcal{O}(1)$ layers for initialization.
- **Newton Iteration**: each exact Newton's iteration requires $\mathcal{O}(1)$ layer. Implementing k iterations requires $\mathcal{O}(k)$ layers.
- **Implementing $\hat{\boldsymbol{w}}_\ell^{\text{TF}}$**: We need one operation of `mov` and `mul` each, requiring $\mathcal{O}(1)$ layer each. Apply Lemma 1 for summation also requires $\mathcal{O}(1)$ layer.
- **Making prediction on test query**: We need one operation of `mov` and `mul` each, requiring $\mathcal{O}(1)$ layer each.

Hence, in total, Transformers can implement k-step Iterative Newton and make predictions accordingly using $\mathcal{O}(k)$ layers.

□

C.3 Iterative Newton as a sum of moments method

Recall that Iterative Newton's method finds $\boldsymbol{S}^\dagger$ as follows

$$\boldsymbol{M}_0 = \underbrace{\frac{2}{\|\boldsymbol{S}\boldsymbol{S}^\top\|_2}}_{\alpha}\boldsymbol{S}^\top, \qquad \boldsymbol{M}_k = 2\boldsymbol{M}_{k-1} - \boldsymbol{M}_{k-1}\boldsymbol{S}\boldsymbol{M}_{k-1}, \forall k \geq 1. \tag{35}$$

We can expand the iterative equation to moments of $\boldsymbol{S}$ as follows.

$$\boldsymbol{M}_1 = 2\boldsymbol{M}_0 - \boldsymbol{M}_0\boldsymbol{S}\boldsymbol{M}_0 = 2\alpha\boldsymbol{S}^\top - 4\alpha^2\boldsymbol{S}^\top\boldsymbol{S}\boldsymbol{S}^\top = 2\alpha\boldsymbol{S} - 4\alpha^2\boldsymbol{S}^3. \tag{36}$$

Let's do this one more time for $\boldsymbol{M}_2$.

$$\begin{aligned}\boldsymbol{M}_2 &= 2\boldsymbol{M}_1 - \boldsymbol{M}_1\boldsymbol{S}\boldsymbol{M}_1 \\ &= 2(2\alpha\boldsymbol{S} - 4\alpha^2\boldsymbol{S}^3) - (2\alpha\boldsymbol{S} - 4\alpha^2\boldsymbol{S}^3)\boldsymbol{S}(2\alpha\boldsymbol{S} - 4\alpha^2\boldsymbol{S}^3) \\ &= 4\alpha\boldsymbol{S} - 8\alpha^2\boldsymbol{S}^3 - 4\alpha^2\boldsymbol{S}^3 + 16\alpha^3\boldsymbol{S}^5 - 16\alpha^4\boldsymbol{S}^7 \\ &= 4\alpha\boldsymbol{S} - 12\alpha^2\boldsymbol{S}^3 + 16\alpha^3\boldsymbol{S}^5 - 16\alpha^4\boldsymbol{S}^7.\end{aligned} \tag{37}$$

We can see that $\boldsymbol{M}_k$ are summations of moments of $\boldsymbol{S}$, with respect to some pre-defined coefficients from the Newton's algorithm. Hence Iterative Newton is a special of an algorithm which computes an approximation of the inverse using higher-order moments of the matrix,

$$\boldsymbol{M}_k = \sum_{s=1}^{2^{k+1}-1} \beta_s \boldsymbol{S}^s \tag{38}$$

with coefficients $\beta_s \in \mathbb{R}$.

We note that Transformer circuits can represent other sum of moments other than Newton's method. We can introduce different coefficients β_i than in the proof of Theorem 1 by scaling the value matrices or through the MLP layers.

C.4 Estimated weight vectors lie in the span of previous examples

What properties can we infer and verify for the weight vectors which arise from Newton's method? A straightforward one arises from interpreting any sum of moments method as a kernel-like method.

We can expand $\boldsymbol{S}^s$ as follows

$$\boldsymbol{S}^s = \left(\sum_{i=1}^t \boldsymbol{x}_i\boldsymbol{x}_i^\top\right)^s = \sum_{i=1}^t \left(\sum_{j_1,\cdots,j_{s-1}} \langle \boldsymbol{x}_i, \boldsymbol{x}_{j_1}\rangle \prod_{v=1}^{s-2} \langle \boldsymbol{x}_{j_v}, \boldsymbol{x}_{j_{v+1}}\rangle\right) \boldsymbol{x}_i\boldsymbol{x}_{j_{s-1}}^\top. \tag{39}$$

Then we have

$$\begin{aligned}\hat{\boldsymbol{w}}_t = \boldsymbol{M}_t\boldsymbol{X}^\top\boldsymbol{y} &= \sum_{s=1}^{2^{t+1}-1} \beta_s \boldsymbol{S}^s\boldsymbol{X}^\top\boldsymbol{y} \\ &= \sum_{s=1}^{2^{t+1}-1} \beta_s \left\{\sum_{i=1}^t \left(\sum_{j_1,\cdots,j_{s-1}} \langle \boldsymbol{x}_i, \boldsymbol{x}_{j_1}\rangle \prod_{v=1}^{s-2} \langle \boldsymbol{x}_{j_v}, \boldsymbol{x}_{j_{v+1}}\rangle\right) \boldsymbol{x}_i\boldsymbol{x}_{j_{s-1}}^\top\right\} \left\{\sum_{i=1}^t y_i\boldsymbol{x}_i\right\} \\ &= \sum_{s=1}^{2^{t+1}-1} \beta_s \left(\sum_{i=1}^t \left(\sum_{j_1,\cdots,j_s} y_{j_s} \langle \boldsymbol{x}_i, \boldsymbol{x}_{j_1}\rangle \prod_{v=1}^{s-1} \langle \boldsymbol{x}_{j_v}, \boldsymbol{x}_{j_{v+1}}\rangle\right) \boldsymbol{x}_i\right) \\ &= \sum_{i=1}^t \underbrace{\left(\sum_{s=1}^{2^{t+1}-1} \beta_s \sum_{j_1,\cdots,j_s} y_{j_s} \langle \boldsymbol{x}_i, \boldsymbol{x}_{j_1}\rangle \prod_{v=1}^{s-1} \langle \boldsymbol{x}_{j_v}, \boldsymbol{x}_{j_{v+1}}\rangle\right)}_{\phi_t(i|\boldsymbol{X},\boldsymbol{y},\boldsymbol{\beta})} \boldsymbol{x}_i \\ &= \sum_{i=1}^t \phi_t(i \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\beta})\, \boldsymbol{x}_i\end{aligned} \tag{40}$$

where $\boldsymbol{X}$ is the data matrix, $\boldsymbol{\beta}$ are coefficients of moments given by the sum of moments method and $\phi_t(\cdot)$ is some function which assigns some weight to the i-th datapoint, based on all other datapoints. Therefore if the Transformer implements a sum of moments method (such as Newton's method), then its induced weight vector $\tilde{\boldsymbol{w}}_t(\text{Transformers} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t)$ after seeing in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^t$ should lie in the span of the examples $\{\boldsymbol{x}_i\}_{i=1}^t$:

$$\tilde{\boldsymbol{w}}_t(\text{Transformers} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t) \stackrel{?}{=} \text{Span}\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_t\} = \sum_{t=1}^t a_i\boldsymbol{x}_i \qquad \text{for coefficients } a_i. \tag{41}$$

We test this hypothesis. Given a sequence of in-context examples $\{\boldsymbol{x}_i, y_i\}_{i=1}^t$, we fit coefficients $\{a_i\}_{i=1}^t$ in Equation (41) to minimize MSE loss:

$$\{\hat{a}_i\}_{i=1}^t = \underset{a_1,a_2,\cdots,a_t\in\mathbb{R}}{\arg\min} \left\| \tilde{\boldsymbol{w}}_t(\text{Transformers} \mid \{\boldsymbol{x}_i, y_i\}_{i=1}^t) - \sum_{t=1}^{t} a_i \boldsymbol{x}_i \right\|_2^2. \tag{42}$$

We then measure the quality of this fit across different number of in-context examples t, and visualize the residual error in Figure 15. We find that even when $t < d$, Transformers' induced weights still lie close to the span of the observed examples $\boldsymbol{x}_i$'s. This provides an additional validation of our proposed mechanism.

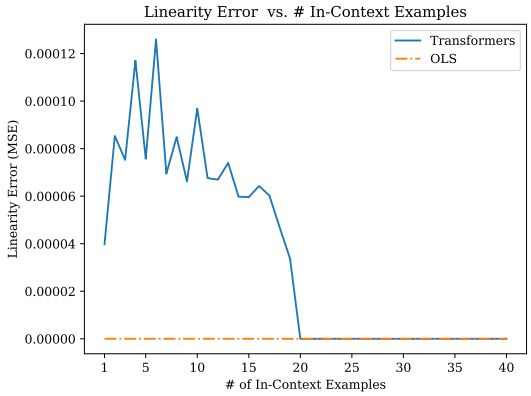


Figure 15: Verification of hypothesis that the Transformers induced weight vector $\boldsymbol{w}$ lies in the span of observed examples $\{\boldsymbol{x}_i\}$.

D Related Work and Discussions

In-context learning by large language models. GPT-3 [6] first showed that Transformer-based large language models can "learn" to perform new tasks from in-context demonstrations (i.e., input-output pairs). Since then, a large body of work in NLP has studied in-context learning, for instance by understanding how the choice and order of demonstrations affects results [20, 19, 33, 39, 8, 26], studying the effect of label noise [23, 49, 47], and proposing methods to improve ICL accuracy [51, 21, 22].

In-context learning beyond natural language. Inspired by the phenomenon of ICL by large language models, subsequent work has studied how Transformers learn in-context beyond NLP tasks. Garg et al. [14] first investigated Transformers' ICL abilities for various classical machine learning problems, including linear regression. We largely adopt their linear regression setup in this work. Li et al. [18] formalize in-context learning as an algorithm learning problem where a Transformer model implicitly constructs a hypothesis function at inference-time and obtain generalization bounds for ICL. Han et al. [15] suggests that Transformers learn in-context by performing Bayesian inference on prompts, which can be asymptotically interpreted as kernel regression. Tarzanagh et al. [40] and Tarzanagh et al. [41] show that Transformers can find max-margin solutions for classification tasks and act as support vector machines. Zhang et al. [50] prove that a linear attention Transformer trained by gradient flow can indeed in-context learn class of linear models. Raventós et al. [32] explore how diverse pretraining data can enable models to perform ICL on new tasks.

Do Transformers implement Gradient Descent? A growing body of work has suggested that Transformers learn in-context by implementing gradient descent within their internal representations. Akyürek et al. [2] summarize operations that Transformers can implement, such as multiplication and affine transformations, and show that Transformers can implement gradient descent for linear regression using these operations. Concurrently, von Oswald et al. [44] argue that Transformers learn in-context via gradient descent, where one layer performs one gradient update. In subsequent work, von Oswald et al. [45] further argue that Transformers are strongly biased towards learning to implement gradient-based optimization routines. Ahn et al. [1] extend the work of von Oswald et al. [44] by showing Transformers can learn to implement preconditioned Gradient Descent, where

the pre-conditioner can adapt to the data. Bai et al. [3] provides detailed constructions for how Transformers can implement a range of learning algorithms via gradient descent. Finally, Dai et al. [11] conduct experiments on NLP tasks and conclude that Transformer-based language models performing ICL behave similarly to models fine-tuned via gradient descent. In this paper, we argue that Transformers actually learn to perform in-context learning by implementing a higher-order optimization method, not gradient descent. Predictions made by different Transformer layers match iterations of higher-order optimization methods better than they match iterations of gradient descent; moreover, Transformers can handle ill-conditioned data, unlike Gradient Descent.

Mechanistic interpretability for Transformers. Our work attempts to understand the mechanism through which Transformers perform in-context learning. Prior work has studied other aspects of Transformers' internal mechanisms, including reverse-engineering language models [46], the grokking phenomenon [29, 25], manipulating attention maps [16], and automated circuit finding [10].

In this work, we studied how Transformers perform in-context learning for linear regression. In contrast with the hypothesis that Transformers learn in-context by implementing gradient descent, our experimental results show that predictions made by different Transformer layers match iterations of Iterative Newton's method better than those of Gradient Descent. Transformers can perform well empirically on ill-conditioned linear regression, whereas first-order methods such as Gradient Descent struggle. Theoretically, we provide exact Transformer circuits that can implement k-steps of Iterative Newton's method with $\mathcal{O}(k)$ layers to make predictions from in-context examples.

An interesting direction is to explore a wider range of higher-order methods that Transformers can implement. It also seems promising to extend our analysis to classification problems, especially given recent work showing that Transformers resemble SVMs in classification tasks [18, 40]. Finally, a natural question is to understand the differences in the model architecture that make Transformers better in-context learners than LSTMs. Based on our investigations with LSTMs, we hypothesize that Transformers can implement more powerful algorithms because of having access to a longer history of examples. Investigating the role of this additional memory in learning appears to be an intriguing direction.