# TOWARDS LEARNING HIGH-PRECISION LEAST SQUARES ALGORITHMS WITH SEQUENCE MODELS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

This paper investigates whether sequence models can learn to perform numerical algorithms, e.g. gradient descent, on the fundamental problem of least squares. Our goal is to inherit two properties of standard algorithms from numerical analysis: (1) *machine precision*, i.e. we want to obtain solutions that are accurate to near floating point error, and (2) *numerical generality*, i.e. we want them to apply broadly across problem instances. We find that prior approaches using Transformers fail to meet these criteria, and identify limitations present in existing architectures and training procedures. First, we show that softmax Transformers struggle to perform high-precision multiplications, which prevents them from precisely learning numerical algorithms. Second, we identify an alternate class of architectures, comprised entirely of polynomials, that can efficiently represent high-precision gradient descent iterates. Finally, we investigate precision bottlenecks during training and address them via a high-precision training recipe that reduces stochastic gradient noise. Our recipe enables us to train two polynomial architectures, gated convolutions and linear attention, to perform gradient descent iterates on least squares problems. For the first time, we demonstrate the ability to train to near *machine precision*. Applied iteratively, our models obtain $100,000\times$ lower MSE than standard Transformers trained end-to-end and they incur a $10,000\times$ smaller generalization gap on out-of-distribution problems. We make progress towards end-to-end learning of numerical algorithms for least squares.

## 1 INTRODUCTION

Least squares is the workhorse of modern numerics: it is well understood theoretically (Boyd & Vandenberghe, 2004; Trefethen & Bau, 2022) and has important downstream applications in science and engineering, including solving regression problems and differential equations (Orszag, 1972; Trefethen, 2000). Thus, least squares has gained interest as a natural testbed for investigating how well ML models can learn to implement algorithms (Garg et al., 2022; Von Oswald et al., 2023).

A surge of recent work suggests that Transformers (Vaswani et al., 2017) can learn to solve least squares using *optimization algorithms* like gradient descent and Newton's method (Akyürek et al., 2022; Fu et al., 2023; Ahn et al., 2024; Bai et al., 2024; Zhang et al., 2023b). These arguments rest on two observations: (1) simplified Transformer architectures (e.g. non-causal linear attention) can exactly implement such algorithms; (2) standard (softmax attention) Transformers learn solutions with similar properties (e.g. convergence rates) as iterative algorithms. Crucially, these works focus on *statistical* least squares: they evaluate Transformer solutions in underdetermined/noisy settings and compare to Bayes-optimal estimators. However, scientific applications like climate or fluids modeling require *numerically precise* solutions to least squares, e.g. to accurately model turbulence or to maintain stable temporal rollouts (Frisch, 1995; Wilcox, 2006). Prior works do not engage with the issue of high precision, so it is still unclear how well Transformers can solve least squares from this perspective.

In this work, we thus study whether existing approaches can solve *numerical* least squares. Specifically, numerical analysis requires that algorithms exhibit (1) *machine precision*, i.e. they should obtain solutions that are accurate to near floating point error, and (2) *numerical generality*, i.e. they are computational procedures that should apply broadly across problem instances. (See Section 2.1 for details.) Since traditional least squares algorithms (e.g. gradient descent and conjugate gradients)

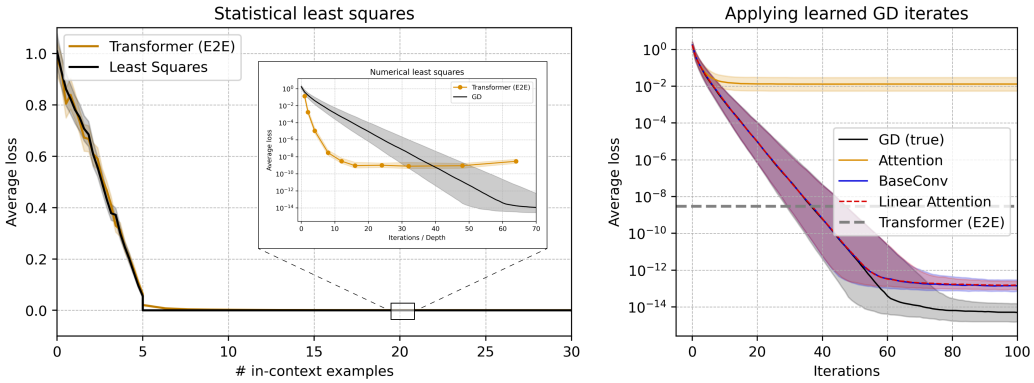


Figure 1: Prior work focuses on *statistical* least squares: Transformers approximate Bayes-optimal estimators (left, adapted from Garg et al. (2022)). In this work, we focus on *numerical* least squares: Transformers struggle to obtain precise solutions (inset). Using a high-precision training recipe, we train two polynomial architectures, BASECONV and linear attention, to perform high-precision gradient descent iterates on least squares (right): applied iteratively, they reach $\approx 10^{-13}$ MSE.

provably meet these criteria (Trefethen & Bau, 2022), it is crucial to evaluate machine learning methods against these same standards to determine their ability to learn numerical algorithms.

We focus on learning the gradient descent (GD) algorithm for least squares. Our study has three parts:

- **We benchmark standard Transformers for precision/generality and identify an expressivity gap on least squares.** When we replicate the standard end-to-end training setup for least squares with Transformers, we find that solutions do not exhibit machine precision and numerical generality (Figure 1a, 2). We identify high-precision multiplications as a fundamental challenge for softmax Transformers. Empirically, on a synthetic element-wise multiplication task, we find precision scales poorly with larger Transformers: an 8-layer model trains to an MSE that is still *10 million* times worse than machine epsilon (Figure 3). Theoretically, we argue that a single layer of softmax attention is unable to exactly express element-wise multiplications. Since implementing GD involves high-precision multiplications, this observation suggests standard Transformers are unable to even precisely *express* GD, much less precisely *learn* the algorithm.

- **We identify an alternate architecture class which does not suffer from expressivity problems.** Motivated by the expressivity limitations of softmax attention, we investigate alternate sequence mixer architectures. Prior work notes that non-causal linear attention is able to exactly implement algorithms like GD and Newton's method (Von Oswald et al., 2023; Giannou et al., 2024) because it consists entirely of *polynomials*. We provide a unified framework to understand existing expressivity results from the lens of arithmetic circuits. In our work, we focus on BASECONV, a gated convolutional architecture, as a case study, since it is provably equivalent to the entire class of polynomial architectures (Arora et al., 2023; 2024). We demonstrate that gated convolutions can express a high-precision GD algorithm ($\approx 10^{-13}$ MSE when implemented in practice, Figure 4).

- **We identify an optimization precision bottleneck and propose a high-precision training recipe.** Although polynomial architectures can precisely express the GD algorithm, we find that standard training procedures struggle to find a solution with sufficiently high precision ($10^{-5}$ MSE, Figure 9). Therefore, towards disentangling precision bottlenecks during training, we first focus on the intermediate task of explicitly learning GD iterates. We identify stochastic gradient noise from minibatching as the main optimization bottleneck, and we find that a simple metric, cosine similarity of minibatch gradients (Liu et al., 2023b), is diagnostic of precision saturation. Towards reducing stochasticity, we propose (1) a learning rate (LR) scheduler that adaptively adjusts LR based on the cosine similarity metric, and (2) to apply EMA over optimizer updates to maintain strong gradient signal. Our high-precision training recipe allows us to train ML architectures to near *machine precision* for the first time. We successfully train two 3-layer models, with BASECONV and linear attention, that learn to perform a single high-precision iteration of GD (Figure 1b). Excitingly, we can also learn multiple GD iterates at once, scaling up to 4 iterations with $10^{-10}$ MSE.

Overall, our work makes the following contributions: (1) we specify the desiderata of learning numerical algorithms, *machine precision* and *numerical generality*, and we demonstrate that standard Transformers fall short because of expressivity limitations of softmax attention; (2) we provide a unified framework using arithmetic circuits to investigate the expressivity of the class of polynomial architectures; (3) we address additional precision bottlenecks that emerge *during training*, even when using expressive polynomial architectures. Although we do not achieve end-to-end learning of GD, we make significant headway: we propose a high-precision training recipe, which, for the first time, allows us to learn iterates of the GD algorithm to near *machine precision*.

## 2 Learning numerical algorithms for least squares

In this section, we distinguish between statistical vs. numerical least squares and discuss the two properties we want our models to inherit from numerical algorithms: *machine precision* and *numerical generality*. We then briefly discuss prior work and, in doing so, tease apart two increasingly end-to-end notions of performing algorithms with ML: *expressing an algorithm in-weights* and *learning algorithm iterates*.

### 2.1 Problem formulation and related work

In this work, our goal is to train a model that solves least squares problems: find $\boldsymbol{x} \in \mathbb{R}^D$ given $\boldsymbol{A} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{b} \in \mathbb{R}^N$ such that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. Here, we briefly discuss two different perspectives on least squares: statistical (in the form of *in-context learning*) and numerical.

**Statistical least squares.** Originally motivated by applications in language modeling, prior works on solving least squares with Transformers typically take a *statistical* perspective. Transformers are trained using an *in-context learning* setup (Garg et al., 2022; Akyürek et al., 2022): problem instances $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ are sampled from a pre-specified distribution $\mathcal{D}_{train}$, and the model is trained to minimize mean squared error (MSE) over $\mathcal{D}_{train}$. Trained models are then evaluated on unseen problem instances, both in and out-of-distribution, and their performance is compared to Bayes-optimal estimators (Garg et al., 2022; Akyürek et al., 2022). We define the in-context least squares training setup in Appendix B and leave a more detailed discussion of related in-context learning work to Appendix A.

**Numerical least squares.** In this work, we instead take a *numerical* perspective on least squares. A prototypical numerical algorithm for least squares is GD. For a problem instance $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, we initialize $\boldsymbol{x}_0$, an estimate of $\boldsymbol{x}$, and iteratively improve our estimate via

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \eta \nabla \mathcal{L}(\boldsymbol{x}_i), \tag{1}$$

where $\mathcal{L}(\hat{\boldsymbol{x}}) := \frac{1}{2}||\boldsymbol{A}\hat{\boldsymbol{x}} - \boldsymbol{b}||_2^2$ is the squared residual error. GD exhibits two properties of numerical algorithms that we want our models to inherit:

- *Machine precision.* Numerical algorithms provably obtain high-precision solutions. For GD, obtaining higher precision simply requires performing more iterations until convergence to machine precision (i.e. the smallest achievable error with floating-point arithmetic) (see Chapter 11 of Trefethen & Bau (2022)). In this work, we use `float32` throughout, where machine precision is $2^{-23} \approx 1.19 \times 10^{-7}$, so we hope for MSEs around $2^{-46} \approx 1.42 \times 10^{-14}$.
- *Numerical generality.* Although the convergence rate of GD depends on the spectrum of $\boldsymbol{A}$ (see Chapter 9 of Boyd & Vandenberghe (2004)), the computational procedure comprising GD is general and can be applied broadly to problem instances. This is unlike statistical generalization and notions of in vs. out-of-distribution. In this work, we are interested to study how closely ML models can emulate the numerical generality of algorithms despite training on a data distribution.

### 2.2 Outline of this work

A recent line of work probes the estimators learned by Transformers on in-context least squares, and suggests that Transformers learn to solve least squares by mimicking iterative algorithms like gradient descent and Newton's method (Von Oswald et al., 2023; Ahn et al., 2024; Fu et al., 2023;

Giannou et al., 2024). These works typically analyze simplified models theoretically and extrapolate to standard training regimes, backed by empirical observations:

- **Theoretical results for simplified models**, e.g. *non-causal linear attention* can implement GD using a specific choice of model weights.
- **Empirical experiments training standard Transformers**, e.g. decoder-only softmax attention Transformers trained *end-to-end* on in-context least squares display convergence rates reminiscent of iterative algorithms.

Although prior works suggest that trained Transformers learn to solve least squares with algorithms, it is still unclear whether statements about learning algorithms in simplified settings transfer to standard Transformers trained end-to-end. We note two significant gaps between previously-analyzed settings and standard in-context least squares:

- **Architectural differences.** Standard Transformers use softmax instead of linear attention, causal instead of non-causal sequence mixers, and include MLPs and LayerNorms (Ba et al., 2016).
- **Optimization.** Even if a model can express a precise and general algorithm, it is unclear whether the model can learn the algorithm from data.

In this work, we tease apart bottlenecks caused by architecture expressivity limitations (Sections 3.3, 4) and optimization difficulties (Section 5) by investigating two increasingly sophisticated notions of performing GD for least squares with ML: *expressing GD in-weights* and *learning GD iterates*.

## 3 Transformers do not learn numerical algorithms in-context

In this section, we evaluate standard Transformers, trained end-to-end, on the criteria of *machine precision* and *numerical generality*. Surprisingly, we demonstrate that existing approaches fail to exhibit these properties: the precision of Transformer solutions saturates $10^6\times$ worse than machine precision (Section 3.1), and their performance further degrades as problem instances deviate from the model's training distribution (Section 3.2). These results suggest that Transformers are not learning proper algorithms as numerical analysis defines them.

Towards identifying expressivity bottlenecks, we identify three linear algebra primitives that comprise standard algorithms including GD and Newton's method (Section 3.3). We find empirically that Transformers struggle to implement high-precision multiplication, and theoretically we argue that softmax attention faces an expressivity gap when trying to exactly express multiplications.

### 3.1 Transformers struggle to reach machine precision

Recent work (Von Oswald et al., 2023; Ahn et al., 2024; Fu et al., 2023; Giannou et al., 2024) studying in-context least squares suggests that Transformers learn to mimic iterative algorithms like GD and Newton's method. Note that if Transformers are able to implement iterative algorithms, the depth of the model should correspond to the number of iterations performed. We thus focus on the simplest case of fully determined least squares problems with fixed size design matrices and investigate whether precision improves as we scale to larger and deeper models.

In Figure 1b, following prior work (Ahn et al., 2024), we fix the size of $\boldsymbol{A} \in \mathbb{R}^{20\times 5}$ and train Transformers end-to-end on least squares, scaling up to $L = 64$ layers. We compare their precision to the convergence rate of the full-batch gradient descent algorithm on least squares. For more details about the training setup, refer to Appendix B.3.1.

At first, Transformer precision scaling exceeds the convergence rate of gradient descent: this finding mirrors similar results reported by Fu et al. (2023), who suggest Transformers may instead be learning higher-order algorithms like Newton's method. However, we further observe that the precision gains for Transformers *rapidly diminish*, such that we observe very little difference in precision between $L = 32$ and $L = 64$ layers. The deepest Transformer models we are able to train achieve an MSE around $10^{-8}$. In contrast, gradient descent converges linearly to machine precision, almost $1,000,000\times$ better precision. The diminishing returns of the Transformer precision scaling imply that Transformers are not learning standard numerical algorithms like GD.

### 3.2 TRANSFORMERS DO NOT EXHIBIT THE GENERALITY OF GRADIENT DESCENT

We further investigate whether Transformers learn solutions to least squares that exhibit numerical generality. Recall that models are trained on a predefined distribution of least squares problems, $\mathcal{D}_{train}$. If Transformers learn to solve least squares using a standard numerical algorithm like GD, then we expect the performance of the model should be robust to out-of-distribution inputs.

For GD specifically, the convergence criterion ($0 < \eta < 2/\sigma_{max}^2$) depends on $\sigma_{max}$, the maximum singular value of $\boldsymbol{A}$, and the optimal rate of convergence depends on the condition number of $\boldsymbol{A}$, $\kappa = \sigma_{max}/\sigma_{min}$ (Boyd & Vandenberghe, 2004). Thus we specify our training distribution $\mathcal{D}_{train}$ over least squares problems ($\boldsymbol{A} \in \mathbb{R}^{20\times 5}$, $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x} \in \mathbb{R}^{20}$) as follows. First, as in prior work (Garg et al., 2022), we sample the entries of $\boldsymbol{A}$ and $\boldsymbol{x}$ i.i.d from a standard Gaussian $N(0, 1)$. We then shift and rescale the singular values of $\boldsymbol{A}$ so that $\sigma_{max} = \kappa = 5$. After training a 12-layer Transformer model on the in-context objective, we evaluate our model on out-of-distribution regression targets $\boldsymbol{b}$.

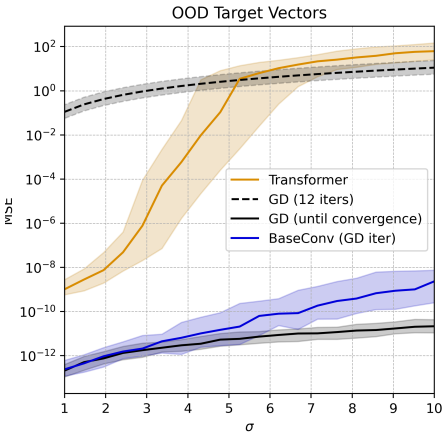


Figure 2: Transformers generalize poorly to out-of-distribution regression targets. In contrast, using our training recipe, we train a BASECONV model to perform high-precision GD iterates. Applied iteratively, our BASECONV model incurs $10,000\times$ less generalization error on out-of-distribution target vectors than the Transformer.

We define $\mathcal{D}^b_{OOD}(\sigma)$ by sampling each entry of $\boldsymbol{x}$ i.i.d. from $N(0, \sigma)$ and computing $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$. Although the distribution of $\boldsymbol{b}$'s and $\boldsymbol{x}$'s changes with $\sigma$, because the spectra of the $\boldsymbol{A}$'s is consistent, we know that GD with fixed choice of $\eta$ will provably converge to high precision.

We find that compared to GD, the Transformer solutions are brittle to unseen regression target distributions. Simply scaling the inputs by a factor of $10\times$, the MSE of the trained Transformer degrades by a factor of $10^8$. In contrast, GD is robust: a fixed number of GD iterations consistently converges to the same order of magnitude of precision (Figure 2). The brittleness of the learned Transformer solution compared to GD again suggests that Transformers are not performing standard numerical algorithms.

### 3.3 IDENTIFYING AN EXPRESSIVITY GAP WITH STANDARD TRANSFORMERS

Toward understanding the limitations of the Transformer architecture, we start with GD and Newton's method, two algorithms used to solve least squares, and look into primitives that comprise them.

**Linear algebra primitives.** We observe that GD and Newton's method can be expressed as compositions of three simple linear algebra operations: sequence-wise read/write (READ), affine transformations (LINEAR), and element-wise multiplications (MULTIPLY). For input $\boldsymbol{u} \in \mathbb{R}^{N\times D}$:

$$\text{READ}(i, j, a, b)(\mathbf{u}) = \begin{cases} \mathbf{u}[k, a{:}b] & k \neq j \\ \mathbf{u}[i, a{:}b] & k = j \end{cases},$$

$$\text{LINEAR}(\boldsymbol{H})(\mathbf{u}) = \mathbf{u}\boldsymbol{H}, \quad \text{where } \boldsymbol{H} : \mathbb{R}^D \to \mathbb{R}^{d_{out}} \text{ is linear,}$$

$$\text{MULTIPLY}(a, b, d_{out})(\mathbf{u}) = \mathbf{u}[:, a{:}a{+}d_{out}] \odot \mathbf{u}[:, b{:}b{+}d_{out}]$$

In Appendix D.2, we define these primitives formally and describe how GD and Newton's method iterates can each be expressed as a composition of these primitives. Intuitively, READ is required to transfer information across the sequence dimension, LINEAR to transfer information across the hidden dimension, and MULTIPLY to compute high-degree interaction terms (like dot products or element-wise squaring).

**Empirical analysis: standard Transformers struggle with multiplication.** We train Transformers on synthetic formulations of these tasks to investigate how precision scales with model size. Details about our training setups are in Appendix B.3.2.

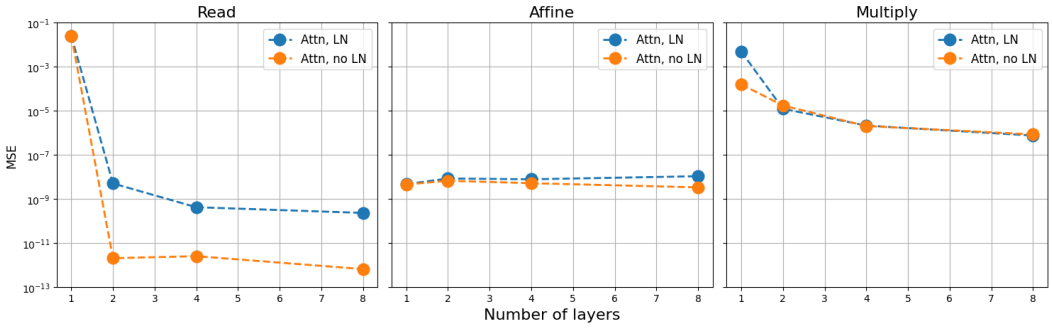


Figure 3: Precision vs. Transformer depth, with and without LayerNorms (LN), on synthetic tasks. While shallow Transformers are able to learn the READ and LINEAR tasks to high precision ($< 10^{-8}$ with 2-layer models), precision on the MULTIPLY task scales poorly with depth (only $10^{-6}$ with 8-layer models).

In Figure 3, we show that even 2-layer Transformers are able to achieve $10^{-8}$ MSE on the READ and LINEAR tasks. However, we find that Transformers struggle with the MULTIPLY task: precision scales poorly with model depth, such that an 8-layer Transformer is only able to achieve $10^{-6}$ MSE. In Appendix C.1, we further show that precision on the MULTIPLY task also scales poorly with increased attention dimension, number of attention heads, and MLP upscaling factor.

**Theoretical analysis: softmax attention struggles to exactly express multiplication.** In Appendix D.2.4, we provide a proof that a single layer of softmax attention cannot exactly express the simple element-wise squaring function $\text{SQUARE}(\boldsymbol{u})[i,j] = \boldsymbol{u}[i,j]^2$ (intuitively, because softmax cannot implement polynomials). Crucially, we note that element-wise squaring is a special case of element-wise multiply, so softmax attention cannot exactly implement MULTIPLY either:

**Theorem 3.1** (Informal statement of Theorem D.31 and Corollary D.32)**.** *One-layer single-headed (causal) softmax attention cannot exactly represent* SQUARE *and* MULTIPLY *for all possible inputs.*

Since precisely implementing numerical algorithms like GD hinges on performing high-precision multiplications, this result suggests that the standard Transformer architecture struggles to precisely implement these algorithms because of a fundamental expressivity gap.

We mention briefly that these findings do not conflict with prior results (Yun et al., 2020b) proving universal approximation theorems for Transformers, because they typically require parameter count to scale exponentially with dimension: see Appendix A.

## 4 ALTERNATE ARCHITECTURES CLOSE THE EXPRESSIVITY GAP

Motivated by the finding that softmax attention struggles to precisely express multiplications, we next investigate alternate sequence mixer architectures. We are inspired by prior results (Von Oswald et al., 2023; Giannou et al., 2024) that show non-causal linear attention is able to exactly implement algorithms like GD and Newton's method. Thus, we focus on the class of *polynomial architectures*, i.e. sequence mixers comprised entirely of polynomial operations, in order to explicitly bake in multiplications. In this section, we present a unified framework that integrates previous findings through the perspective of arithmetic circuits. Specifically, we focus on BASECONV, a gated convolutional model that combines element-wise multiplications (gating) with long convolutions. We work with BASECONV for two reasons:

- Recent work (Arora et al., 2023; 2024) has shown that BASECONV is equivalent to general arithmetic circuits, including all polynomial architectures. Thus, existing results with other polynomial architectures, e.g. linear attention, transfer directly to BASECONV.
- Empirically, gated convolutional models have been shown to perform comparably to attention-based architectures on tasks like language, audio, and DNA modeling (Arora et al., 2024; Nguyen et al., 2024; Zhang et al., 2023a).

We emphasize that although we find gated convolutions are convenient to work with theoretically and empirically, we believe that other sequence mixer architectures may also be able to alleviate the expressivity issues we highlight in Section 3.3. In particular, we show promising empirical results for non-causal linear attention in Section 5.2.

### 4.1 GATED CONVOLUTIONS ARE EQUIVALENT TO ARITHMETIC CIRCUITS

**BASECONV definition.** In this work, we focus on a variant of the BASECONV operator from Arora et al. (2023). Given an input $\boldsymbol{u} \in \mathbb{R}^{N \times D}$, BASECONV$(\boldsymbol{u})$ is defined as:

$$(\underbrace{(\boldsymbol{u}\boldsymbol{W}_{gate} + \boldsymbol{b}_{gate})}_{\textbf{Linear Projection}} \odot \underbrace{(\boldsymbol{h} * (\boldsymbol{u}\boldsymbol{W}_{in} + \boldsymbol{b}_{in}) + \boldsymbol{b}_{conv})}_{\textbf{Convolution}})\boldsymbol{W}_{out} + \boldsymbol{b}_{out} \tag{2}$$

where the layer is parameterized by learnable filters $\boldsymbol{h} \in \mathbb{R}^{N \times D}$, linear projections $\boldsymbol{W}_{in}, \boldsymbol{W}_{gate}, \boldsymbol{W}_{out} \in \mathbb{R}^{D \times D}$ , and bias matrices $\boldsymbol{b}_{conv}, \boldsymbol{b}_{in}, \boldsymbol{b}_{gate}, \boldsymbol{b}_{out} \in \mathbb{R}^{N \times D}$. Here, $\odot$ represents the Hadamard product, and convolution of two matrices is computed as convolution of the corresponding columns.

**BASECONVs can exactly express linear algebra primitives.** In Appendix D.2.1, we provide explicit constructions of single-layer BASECONV models that exactly implement the READ, LINEAR, and MULTIPLY primitives from Section 3.3.

We note that this result is stronger than prior BASECONV expressivity results (e.g. Theorem H.21 from Arora et al. (2023)), which imply a poly-log-factor increase in parameters (specifically layers) translating from arbitrary arithmetic circuits. Here, we show by construction that these specific primitives, and any circuits that are compositions of them, incur only a constant factor loss.

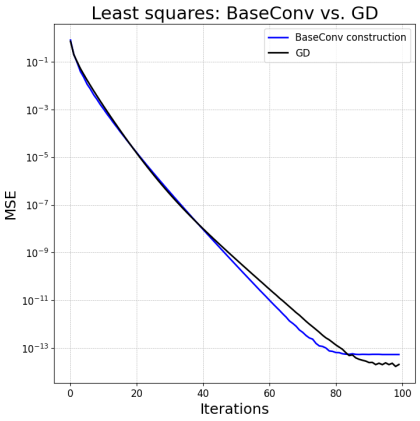


Figure 4: BASECONV can express high-precision gradient descent: our implementation of the weight construction reaches $10^{-13}$ MSE in practice.

**BASECONVs can perfectly recover linear algebra primitives from data.** In Appendix D.7, for SQUARE and LINEAR, we further show the following under mild assumptions, which our input distribution satisfies (see details in Assumptions D.45, D.46, D.55, D.61):

**Theorem 4.1** (Informal statement of Theorems D.59, D.62)**.** BASECONV *perfectly recovers* SQUARE *and* LINEAR *when it achieves zero population gradient w.r.t. MSE loss.*

We note that although results of the form "exact solution implies zero population gradient" exist in the literature (Ahn et al., 2024; Mahankali et al., 2023), to the best of our knowledge, we are the first to show the *converse* ("zero population gradient implies recovery of exact solution") for sequence model architectures. In Appendix C.1, we show that BASECONV models can learn the READ, LINEAR, and MULTIPLY primitives to high precision in practice (Figure 6).

**BASECONVs are universal approximators.** Finally, we show in Appendix D.4 that BASECONV can efficiently approximate *smooth functions* by implementing polynomials:

**Theorem 4.2** (Informal statement of Theorem D.39)**.** *Given a $k$-times differentiable function $\bar{f} : [-1, 1] \to \mathbb{R}$, define $f : [-1, 1]^{N \to D} \to \mathbb{R}^{N \times D}$, which applies $\bar{f}$ element-wise to all inputs. Then $\forall \epsilon > 0$, there exists a* BASECONV *model approximates $f$ to within error $\epsilon$, with $O\left(\sqrt[k]{\frac{L}{\epsilon}}\right) + k$ depth and $O(ND)$ parameters, where $||f^{(k)}||_\infty \leq L$.*

We additionally prove a universal approximation theorem for general smooth multivariate functions in Appendix D.4 (Theorem D.44).

### 4.2 BASECONV CAN PRECISELY EXPRESS GRADIENT DESCENT FOR LEAST SQUARES

We now focus on the gradient descent algorithm for least squares. Explicitly, given a least squares problem instance $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and an initial iterate $\boldsymbol{x}_0$, a single iteration of gradient descent computes

$$\boldsymbol{x}_1 := \boldsymbol{x}_0 - \eta \nabla \mathcal{L}(\boldsymbol{x}_0), \text{ where } \nabla_{\boldsymbol{x}} \mathcal{L} = \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}). \tag{3}$$

We provide two explicit $O(1)$-layer weight constructions to express a GD iterate using BASECONV in Appendix D.3.1. One requires a $O(D)$ state size using a *non-causal* model (i.e. each entry can access

any other entry of the sequence) and one requires a $O(D^2)$ state size using a *causal* model (i.e. entries cannot access later entries of the sequence). In Appendix D.3.2, we prove that both constructions are asymptotically optimal with respect to state size.

In Figure 4, we implement our non-causal weight construction into a deep BASECONV model as a proof of concept. We confirm that gated convolutions can empirically implement high-precision gradient descent – notably, roundoff errors due to machine precision do not significantly accumulate in practice, despite scaling up to a depth-100 BASECONV model.

## 5 TOWARDS TRAINING MODELS TO MACHINE PRECISION

Although BASECONVs are expressive enough to solve least squares precisely, we find that simply swapping out softmax attention with BASECONV and training end-to-end is insufficient for high precision: our BASECONV models perform as poorly as standard Transformers (Figure 9). This suggests that additional precision bottlenecks are present *during high-precision training*. In this section, we thus investigate what it takes to train polynomial architectures to machine precision.

Recent works (Rodionov & Prokhorenkova, 2023; 2024) on algorithm learning find that intermediate supervision is crucial for learning long computation trajectories. We hypothesize that end-to-end least squares faces a similar challenge. Thus, to study high-precision optimization, we first investigate a simplified setting: learning to perform *explicit GD updates* for least squares.

Using this task as a benchmark, we identify a fundamental bottleneck in high-precision regimes, *gradient variance from minibatching*, and we identify a metric based on cosine similarity of successive gradients that is diagnostic of precision saturation during training. We then propose a high-precision training recipe, which for the first time allows us to train ML models to near *machine precision*. Using our training recipe, we learn to perform explicit GD updates to $10^{-13}$ average MSE (Figure 1b), and we can also learn up to 4 iterates of GD at once with an MSE of $10^{-10}$ (Table 7).

**Simplifying the training setup.** We first define a sequence of *$k$-th iterate* tasks, where the goal is to explicitly produce the $k$-th iterate of GD given a least squares problem instance $(\boldsymbol{A}, \boldsymbol{b})$, an initial iterate $\boldsymbol{x}_0$, and a step size $\eta$:

$$\{(\boldsymbol{a}_1, b_1), \ldots, (\boldsymbol{a}_N, b_N), \boldsymbol{x}_0\} \to \boldsymbol{x}_k, \text{ where } \boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \eta \nabla \mathcal{L}(\boldsymbol{x}_i),\ i \in [k-1]. \tag{4}$$

We then define the *explicit gradient* task, where the goal is to produce the GD update vector:

$$\{(\boldsymbol{a}_1, b_1), \ldots, (\boldsymbol{a}_N, b_N), \boldsymbol{x}_0\} \to \nabla \mathcal{L}(\boldsymbol{x}_0). \tag{5}$$

Note that (up to a residual connection), the explicit gradient task is equivalent to 1-step GD, and standard in-context least squares is equivalent to taking $k \to \infty$. Thus, the explicit gradient task is a natural simplification of standard in-context least squares, and the $k$-th iterate task allows us to smoothly interpolate between the two extremes of difficulty. Refer to Appendix B for more details.

### 5.1 TOWARDS A HIGH-PRECISION TRAINING RECIPE

Our theoretical results in Section 4 imply that a 3-layer BASECONV is expressive enough to solve the explicit gradient task, so we use training a 3-layer BASECONV on this task as our benchmark for studying the challenges of high-precision learning.

**Precision saturates with standard training procedures.** Motivated by prior work (Garg et al., 2022; Von Oswald et al., 2023; Ahn et al., 2024), we start by investigating two basic optimization procedures: Adam with constant learning rate (LR) and with exponentially decaying LR.

In Appendix C.2 (Figure 10), we sweep initial LR and LR steprate across 2-3 orders of magnitude for constant and decaying LR schedules. We find:

- *Precision saturation occurs with both constant and decaying LR schedules.* After a number of training iterations, the average loss saturates and is unable to improve. We note that this occurs even while gradients magnitudes and LR are non-zero.
- *Slower-decaying LR schedules perform better but require exponentially more training iterations.* In Figure 8, we further analyze this phenomenon in the simpler case of 1-layer Transformers/BASECONVs on the MULTIPLY synthetic. We observe a power-law relation between precision

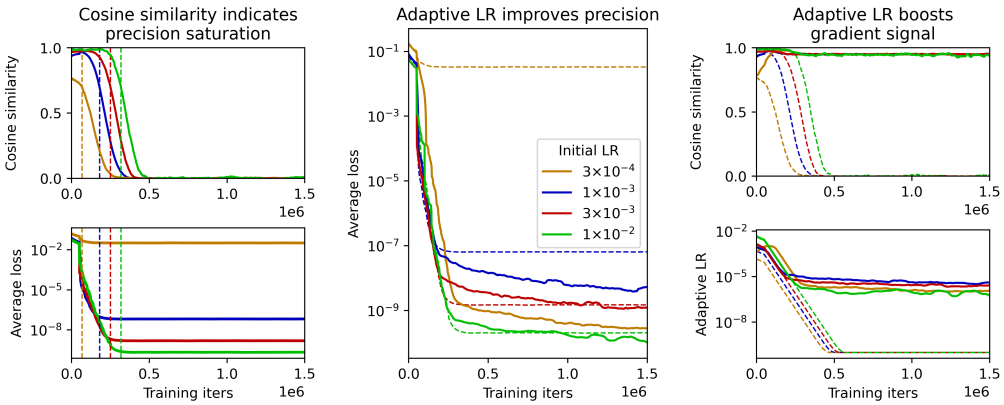


Figure 5: Gradient metric is predictive of precision saturation (left). We propose a simple adaptive LR scheduler that alleviates precision saturation (middle). Adaptive LR effectively boosts gradient signal during training (right).

and number of training iterations as we sweep steprate; although it may be possible to train to high precision in theory, this approach seems infeasible in practice.

- *With aggressively-decaying LR schedules, higher initial LR is better.* For a fixed scheduler step rate, increasing initial LR leads to significant improvements in final MSE, e.g. in Figure 10, an improvement of $1000\times$ simply by increasing initial LR from $10^{-3}$ to $10^{-2}$. Choosing a LR that is too large causes training instability, so in practice we set it to the *largest value that trains stably*.

Our analysis suggests that Adam with an exponentially decreasing LR scheduler gets us only part of the way to a machine precision training recipe. We next address the issue of precision saturation.

**Stochastic gradients bottleneck precision.** We identify *minibatch gradient variance* as the main source of precision saturation. Although our goal is to minimize the expected loss over problem instances from $\mathcal{D}_{train}$, in practice we minimize over finite minibatch samples instead. Minibatch training is the standard in ML, but interestingly we find that the variance in minibatch gradients can dominate the population gradient signal in high-precision regimes, causing the loss to stagnate.

To demonstrate this, we define a simple metric to assess the strength of the gradient signal during training. At a given training step, we take the current model weights, sample $n$ different minibatches of least squares problems, and compute the minibatch model gradients $\{g_1, \ldots, g_n\}$. We then compute the *average cosine similarity* between all pairs, as in Liu et al. (2023b):

$$\sigma_g := \frac{2}{n(n-1)} \sum_{i \neq j} \frac{g_i^T g_j}{||g_i||_2 ||g_j||_2} \tag{6}$$

We observe that this cosine similarity metric is predictive of precision saturation across MSE scales and optimizer hyperparameters (Figure 5).

**An adaptive LR scheduler boosts gradient signal beyond precision saturation.** We thus propose an *adaptive* LR scheduler based on the gradient variance. Our scheduler is motivated by two intuitions:

- Whenever the cosine similarity metric is *high*, gradient signal is strong. In order to refine the highest-precision bits of the model weights, we need to slowly *decrease* the LR.
- Whenever the cosine similarity metric is *low*, the model weights are stuck in a local region of the loss landscape. To allow the model to escape this region, we need to *increase* the LR.

The basic scheduler we use in this work simply decreases the LR exponentially while the metric is above a threshold $\sigma_{th}$ and increases the LR instead if the metric is below $\sigma_{th}$. In Figure 5, we show that this simple approach alleviates the loss saturation phenomenon: we see a boost in population loss across our LR settings, and we observe the models consistently improve as we continue training. We note that proper choice of LR hyperparameters is still crucial for *efficient* convergence to machine precision – we leave speeding up the convergence rate via better adaptive schedulers to future work.

**Exponential Moving Average (EMA) over optimizer updates.** Finally, motivated by our observation that gradient variance bottlenecks precision and inspired by recent works (Lee et al., 2024; Pagliardini et al., 2024), we apply an additional EMA over Adam's update vectors to help smooth out minibatch noise. Empirically, we find this boosts the final MSE by as much as $100,000\times$ on the explicit gradient task: see Appendix C.2 (Figure 11).

Our training recipe for efficient high-precision convergence thus involves two techniques: (1) an adaptive LR scheduler that exponentially increases or decays LR according to the cosine similarity metric; and (2) applying EMA over optimizer updates.

### 5.2 Learning high-precision gradient descent with polynomial architectures

Using our training recipe, we successfully train two 3-layer models with polynomial architectures, BaseConv and non-causal linear attention, on the explicit gradient task. For the first time, we are able to train to near *machine precision*: we achieve an average loss of $10^{-13}$ MSE.

In Figure 1b, we slot our trained models into the standard GD algorithm, using their predictions in place of the true least squares gradients $\nabla\mathcal{L}$. Specifically, for a least squares problem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and initial iterate $\boldsymbol{x}_0$, we repeatedly compute $\boldsymbol{x}_{i+1} := \boldsymbol{x}_i - \eta\boldsymbol{\Delta}_i$, where $\boldsymbol{\Delta}_i := T_\theta(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}_i)$ is the prediction of the model. We iteratively apply the model until convergence to a fixed point $\boldsymbol{x}_\infty$.

We find that both our models achieve high precision. In this setting, we reach an average MSE of $10^{-12}$ (Figure 1, right): this is $100,000\times$ better MSE than the biggest Transformers we are able to train end-to-end. Moreover, our BaseConv model exhibits better numerical generality than the Transformer, incurring a $10,000\times$ smaller generalization gap on problems outside its training distribution (Figure 2). Interestingly, we find that our linear attention model exhibits markedly worse generality: its out-of-distribution performance nearly matches the Transformer's, and the model iterates eventually diverge: see Figure 13.

**Learning $k$-iterates of GD for larger $k$.** We find that our training recipe also allows us to learn up to $k = 4$ iterates of GD at once with $10^{-10}$ MSE: see Table 7 and Figure 14 for results. We are not able to stably train deeper models without reintroducing non-polynomial normalization techniques like LayerNorms, which causes precision bottlenecks. For small $k$, we observe that LayerNorms worsen precision by over $1,000\times$. See Appendix C.3 for details.

**Experiments with in-context ODE solving.** Finally, towards high-precision ML for more realistic tasks, we provide preliminary results on in-context ODE solving. We find that our proposed techniques outperform standard Transformers by up to $1,000,000\times$ in MSE (up to $\approx 10^{-10}$ with iterative BaseConvs vs. $\approx 10^{-4}$ with 12-layer Transformers). See Appendix C.4 for details.

## 6 Discussion and Limitations

In this work, we investigate learning to solve least squares from a numerical perspective. We find that Transformers fail to learn solutions that exhibit the properties of machine precision and numerical generality. Disentangling effects from the model architecture and optimizer, we find that standard design choices perform surprisingly poorly from the lens of numerics. We identify expressivity limitations with softmax attention, and find surprisingly that even MLPs and LayerNorms significantly affect precision (up to $1,000,000\times$ worse MSE on the explicit gradients task). On the optimization front, we find stochastic gradient noise from minibatch training becomes a precision bottleneck in high-precision regimes. We propose an adaptive LR scheduler that alleviates this issue on a simplified task, but we suspect that this issue remains a fundamental challenge on harder problems. Crucially, although we make progress toward learning to solve numerical least squares end-to-end, our techniques struggle to maintain stable and precise training with deep networks.

We note that the *numerical* criteria we consider in this work represent a fundamentally different type of learning and generalization from *statistical* notions that are prevalent in ML. We believe these numerical perspectives may be relevant to the wider scientific ML community. For example, existing approaches to solving PDEs have shown promise but are known to be brittle outside their training distributions (Wang & Lai, 2023; Rathore et al., 2024). This inhibits their usefulness in high-impact applications like climate or fluids modeling, where high precision and robustness are crucial. We believe learning to implement precise numerical algorithms directly from data is an exciting prospect that has the potential to unlock new capabilities across science and engineering.

### Reproducibility Statement

We will release the code and configuration files necessary to reproduce our experiments upon acceptance of this paper. In this work, all experiments are done using synthetic data and tasks. All experiments were conducted using PyTorch on NVIDIA A100/H100 GPUs. Detailed hyperparameters (learning rate, batch size, and optimizer settings) and proofs of all theoretical claims are provided in the supplementary materials.

## References


Kwangjun Ahn, Xiang Cheng, Hadi Daneshmand, and Suvrit Sra. Transformers learn to implement preconditioned gradient descent for in-context learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Kabir Ahuja, Madhur Panwar, and Navin Goyal. In-context learning through the bayesian prism. *arXiv preprint arXiv:2306.04891*, 2023.

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Arhitectures and algorithms. *arXiv preprint arXiv:2401.12973*, 2024.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and Improving Recall in Efficient Language Models, 2023.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff, 2024.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. *Advances in neural information processing systems*, 36, 2024.

Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Wuyang Chen, Jialin Song, Pu Ren, Shashank Subramanian, Dmitriy Morozov, and Michael W Mahoney. Data-efficient operator learning via unsupervised pretraining and in-context learning. *arXiv preprint arXiv:2402.15734*, 2024.

Xiang Cheng, Yuxin Chen, and Suvrit Sra. Transformers implement functional gradient descent to learn non-linear functions in context, 2024. URL https://arxiv.org/abs/2312.06528.

David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders. In *International Conference on Machine Learning*, pp. 5544–5562. PMLR, 2023.

Liam Collins, Advait Parulekar, Aryan Mokhtari, Sujay Sanghavi, and Sanjay Shakkottai. In-context learning with transformers: Softmax attention adapts to function lipschitzness, 2024. URL https://arxiv.org/abs/2402.11639.

D. Jackson. *The theory of approximation*. Amer. Math. Soc. Colloq. Publ., vol. 11, Amer. Math. Soc, Providence, R. I., 1930.

Tri Dao, Nimit S Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2020.

Ishita Dasgupta, Andrew K Lampinen, Stephanie CY Chan, Antonia Creswell, Dharshan Kumaran, James L McClelland, and Felix Hill. Language models show human-like content effects on reasoning. *arXiv preprint arXiv:2207.07051*, 2022.

Uriel Frisch. *Turbulence: the legacy of AN Kolmogorov*. Cambridge university press, 1995.

Daniel Y Fu, Tri Dao, Khaled K Saab, Armin W Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

Deqing Fu, Tian-Qi Chen, Robin Jia, and Vatsal Sharan. Transformers learn higher-order optimization methods for in-context learning: A study with linear models. *arXiv preprint arXiv:2310.17086*, 2023.

Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Angeliki Giannou, Liu Yang, Tianhao Wang, Dimitris Papailiopoulos, and Jason D Lee. How well can transformers emulate in-context newton's method? *arXiv preprint arXiv:2403.03183*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Michael T Heideman and C Sidney Burrus. *Multiplicative complexity, convolution, and the DFT*. Springer, 1988.

Maximilian Herde, Bogdan Raonić, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for pdes, 2024. URL https://arxiv.org/abs/2405.19101.

Yu Huang, Yuan Cheng, and Yingbin Liang. In-context convergence of transformers. *arXiv preprint arXiv:2310.05249*, 2023.

Jaerin Lee, Bong Gyun Kang, Kihoon Kim, and Kyoung Mu Lee. Grokfast: Accelerated grokking by amplifying slow gradients, 2024. URL https://arxiv.org/abs/2405.20233.

Jerry Weihong Liu, N Benjamin Erichson, Kush Bhatia, Michael W Mahoney, and Christopher Re. Does in-context operator learning generalize to domain-shifted settings? In *The Symbiosis of Deep Learning and Differential Equations III*, 2023a.

Zhuang Liu, Zhiqiu Xu, Joseph Jin, Zhiqiang Shen, and Trevor Darrell. Dropout reduces underfitting, 2023b. URL https://arxiv.org/abs/2303.01500.

Arvind Mahankali, Tatsunori B Hashimoto, and Tengyu Ma. One step of gradient descent is provably the optimal in-context learner with one layer of linear self-attention. *arXiv preprint arXiv:2307.03576*, 2023.

Nick McGreivy and Ammar Hakim. Weak baselines and reporting biases lead to overoptimism in machine learning for fluid-related partial differential equations. *Nature Machine Intelligence*, 6 (10):1256–1269, September 2024. ISSN 2522-5839. doi: 10.1038/s42256-024-00897-5. URL http://dx.doi.org/10.1038/s42256-024-00897-5.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

William Merrill and Ashish Sabharwals. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023. doi: 10.1162/tacl_a_00562. URL https://aclanthology.org/2023.tacl-1.31.

Eric J. Michaud, Ziming Liu, and Max Tegmark. Precision machine learning. *Entropy*, 25(1):175, January 2023. ISSN 1099-4300. doi: 10.3390/e25010175. URL http://dx.doi.org/10.3390/e25010175.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability, 2023. URL https://arxiv.org/abs/2301.05217.

Eric Nguyen, Michael Poli, Matthew G. Durrant, Armin W. Thomas, Brian Kang, Jeremy Sullivan, Madelena Y. Ng, Ashley Lewis, Aman Patel, Aaron Lou, Stefano Ermon, Stephen A. Baccus, Tina Hernandez-Boussard, Christopher Ré, Patrick D. Hsu, and Brian L. Hie. Sequence modeling and design from molecular to genome scale with evo. *bioRxiv*, 2024. doi: 10.1101/2024.02.27.582234. URL https://www.biorxiv.org/content/early/2024/02/27/2024.02.27.582234.

Steven A Orszag. Comparison of pseudospectral and spectral approximation. *Studies in Applied Mathematics*, 51(3):253–259, 1972.

Matteo Pagliardini, Pierre Ablin, and David Grangier. The ademamix optimizer: Better, faster, older, 2024. URL https://arxiv.org/abs/2409.03137.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Peter Bürgisser and Michael Clausen and M. Amin Shokrollah. *Algebraic Complexity Theory*. Springer, 1997.

Tobias Von Petersdorff. Polynomial approximation and interpolation. 2015. Numerical Analysis Class Notes. https://www.math.umd.edu/~petersd/666/amsc666notes02.pdf.

W. Pleśniak. Multivariate jackson inequality. *Journal of Computational and Applied Mathematics*, 233(3):815–820, 2009. ISSN 0377-0427. doi: https://doi.org/10.1016/j.cam.2009.02.095. URL https://www.sciencedirect.com/science/article/pii/S0377042709001307. 9th OPSFA Conference.

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pp. 28043–28078. PMLR, 2023.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets, 2022. URL https://arxiv.org/abs/2201.02177.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Pratik Rathore, Weimu Lei, Zachary Frangella, Lu Lu, and Madeleine Udell. Challenges in training pinns: A loss landscape perspective, 2024. URL https://arxiv.org/abs/2402.01868.

Allan Raventós, Mansheej Paul, Feng Chen, and Surya Ganguli. Pretraining task diversity and the emergence of non-bayesian in-context learning for regression. *Advances in Neural Information Processing Systems*, 36, 2024.

Gleb Rodionov and Liudmila Prokhorenkova. Neural algorithmic reasoning without intermediate supervision, 2023. URL https://arxiv.org/abs/2306.13411.

Gleb Rodionov and Liudmila Prokhorenkova. Discrete neural algorithmic reasoning, 2024. URL https://arxiv.org/abs/2402.11628.

Günther Schulz. Iterative berechung der reziproken matrix. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 13(1):57–59, 1933.

Smoothness. Smoothness — Wikipedia, the free encyclopedia, 2006. https://en.wikipedia.org/wiki/Smoothness.

Gilbert Strang. *Linear algebra and its applications*. 2012.

Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra, Twenty-fifth Anniversary Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2022. doi: 10.1137/1.9781611977165. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611977165.

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.

Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark, 2022. URL https://arxiv.org/abs/2205.15659.

Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pp. 35151–35174. PMLR, 2023.

Yongji Wang and Ching-Yao Lai. Multi-stage neural networks: Function approximator of machine precision, 2023. URL https://arxiv.org/abs/2307.08934.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

Sanford Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.

D.C. Wilcox. *Turbulence Modeling for CFD*. Number v. 1 in Turbulence Modeling for CFD. DCW Industries, 2006. ISBN 9781928729082. URL https://books.google.com/books?id=tFNNPgAACAAJ.

Steve Yadlowsky, Lyric Doshi, and Nilesh Tripuraneni. Pretraining data mixtures enable narrow model selection capabilities in transformer models. *arXiv preprint arXiv:2311.00871*, 2023.

Liu Yang, Siting Liu, Tingwei Meng, and Stanley J. Osher. In-context operator learning with data prompts for differential equation problems. *Proceedings of the National Academy of Sciences*, 120(39), September 2023a. ISSN 1091-6490. doi: 10.1073/pnas.2310142120. URL http://dx.doi.org/10.1073/pnas.2310142120.

Liu Yang, Siting Liu, Tingwei Meng, and Stanley J Osher. In-context operator learning for differential equation problems. *arXiv preprint arXiv:2304.07993*, 2023b.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions?, 2020a.

Chulhee Yun, Yin-Wen Chang, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. O (n) connections are expressive enough: Universal approximability of sparse transformers. *Advances in Neural Information Processing Systems*, 33:13783–13794, 2020b.

Michael Zhang, Khaled K. Saab, Michael Poli, Tri Dao, Karan Goel, and Christopher Ré. Effectively modeling time series with simple discrete state spaces, 2023a. URL https://arxiv.org/abs/2303.09489.

Ruiqi Zhang, Spencer Frei, and Peter L Bartlett. Trained transformers learn linear models in-context. *arXiv preprint arXiv:2306.09927*, 2023b.

# APPENDIX

The appendix is organized as follows:

- Appendix A provides a more detailed overview of related work.
- Appendix B provides details about our experimental setup.
- Appendix C provides additional experiments and ablation studies.
- Appendix D provides details about our main theoretical results.

# A Extended background

## A.1 Least squares

Least squares, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, is well-understood theoretically, and we know of simple numerical algorithms for solving least squares to high precision (Weisberg, 2005; Boyd & Vandenberghe, 2004). We focus on two algorithms: gradient descent and Newton's method.

**Gradient descent** Given a guess for $\boldsymbol{x}^*$, we minimize the least squares loss

$$\mathcal{L}(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{N}(\boldsymbol{a}_i^T\boldsymbol{x} - b_i)^2 \tag{7}$$

via gradient descent on $\boldsymbol{x}$:

$$\nabla_{\boldsymbol{x}}\mathcal{L}_N = \sum_{i=1}^{N}(\boldsymbol{x}^T\boldsymbol{a}_i - b_i)\boldsymbol{a}_i \tag{8}$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta\nabla\mathcal{L}_N(\boldsymbol{x}_t) \tag{9}$$

**Ordinary Least Squares and Newton's method** In the noiseless, full determined regime, the Bayes-optimal estimator is ordinary least squares (OLS) (Weisberg, 2005):

$$\boldsymbol{x}^{OLS} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}, \tag{10}$$

where

$$\boldsymbol{A} = \begin{pmatrix} \leftarrow \boldsymbol{a}_1 \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{a}_N \rightarrow \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix} \tag{11}$$

Note that this estimator requires a matrix inverse, which is expensive to compute exactly. An alternative is to use Newton's method to approximate the matrix inverse term (Schulz, 1933). To estimate $(\boldsymbol{A}^T\boldsymbol{A})^{-1}$, we can perform the following iterative algorithm:

$$\boldsymbol{M}_{t+1} = \boldsymbol{M}_t(2\boldsymbol{I} - (\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{M}_t) \tag{12}$$

where $\boldsymbol{M}_t$ converges to $(\boldsymbol{A}^T\boldsymbol{A})^{-1}$.

## A.2 Related work

In this section, we detail prior work on in-context learning, Transformer expressivity, gated convolutional architectures, and algorithm learning.

**In-context learning.** The capability of Transformers to perform in-context learning on language and pattern matching tasks has been well-documented (Brown et al., 2020; Dasgupta et al., 2022; Wei et al., 2022). More recently, a flurry of work has investigated in-context learning for regression-style tasks. Garg et al. (2022) first formulated the mathematical framework to analyze the estimators Transformers implement in-context, focusing on linear regression and other least squares problems. A number of works further observed empirically that Transformers seem to approximate Bayes-optimal estimators on distributional problems. For example, based on the task distribution, the performance of in-context Transformers mimics optimally-tuned LASSO on sparse linear regression, ridge regression on noisy dense linear regression, and Bayes-optimal priors for task mixtures (Akyürek et al., 2024; Raventós et al., 2024; Yadlowsky et al., 2023; Ahuja et al., 2023; Bai et al., 2024). Beyond standard least squares problems, other works have investigated the ability of Transformers to in-context solve broader problems of scientific interest like differential equations (Yang et al., 2023b; Chen et al., 2024; Liu et al., 2023a).

Towards explaining these observations, recent works have focused on understanding the expressivity and optimziation landscapes of Transformer variants (typically non-causal linear attention) on linear regression. Linear attention has been shown to be expressive enough to implement numerical algorithms for solving linear regression, including gradient descent (Akyürek et al., 2022; Von Oswald

et al., 2023) and Newton's method (Fu et al., 2023; Giannou et al., 2024). Recent work (Ahn et al., 2024; Mahankali et al., 2023; Zhang et al., 2023b) has also begun to investigate the optimization dynamics for linear attention on least squares. Finally, we highlight that recent work (Bai et al., 2024; Huang et al., 2023; Collins et al., 2024; Cheng et al., 2024) makes progress on theoretically understanding non-linear attention, e.g. with softmax or ReLU activations.

Unlike prior work, we investigate the capabilities of standard Transformers, focusing on exploring their capability to perform *high-precision* optimization algorithms. Noting a gap between empirical performance and theoretical claims regarding in-context least squares as gradient descent, we further investigate alternative architectures to softmax attention.

**Expressivity and approximation ability of Transformers.** Although Transformers were initially designed for discrete tasks like language modeling, recent works have investigated the ability of the Transformer architecture to express general *continuous-valued* sequence-to-sequence maps. We briefly mention three classes of prior work:

- **Constructive arguments.** We highlight Giannou et al. (2023), which proposes a looped-Transformer weight construction that implements a basic mathematical instruction set. Using compositions of these instructions, the authors demonstrate that Transformers are expressive enough to implement numerical algorithms, including matrix inversion and SGD on linear models.
- **Universal approximation results.** Several works, such as Yun et al. (2020a;b), provide bounds on the number of parameters and layers required to approximate smooth sequence-to-sequence functions to arbitrary precision using Transformers. However, these results typically require parameters to scale exponentially with respect to problem size, which quickly becomes impractical in practice.
- **Complexity theory results.** Recent works (Chiang et al., 2023; Merrill & Sabharwals, 2023; Merrill & Sabharwal, 2024) prove that *log-precision* Transformers lie in $TC^0$, a limited complexity class of circuits.

**Gated convolutions.** Gated convolutional models are a class of architectures that serve as an efficient alternative to attention. These models, consisting of gating (element-wise multiplication) and long convolutions (filter size equal to sequence length), stem from earlier work (Gu et al., 2021) inspired by the signal processing literature. In this work we focus on the BASECONV model from Arora et al. (2023), but a recent surge of interest in efficient attention replacements has led to a flood of gated convolutional architectures (Poli et al., 2023; Peng et al., 2023; Gu & Dao, 2023).

Recent architectural innovations within the class of gated convolutional models have been largely motivated by language modeling tasks (Fu et al., 2022; Arora et al., 2023). Unlike these prior works, which focus on matching attention's performance on *discrete* tasks, we observe that the connection between gated convolutions and arithmetic circuits implies they are able to exactly express a range of important numerical algorithms for *continuous-valued* tasks. We further investigate their ability to learn these algorithms in-context.

**Algorithm learning.** We mention two lines of work related to learning algorithms using ML:

- **Grokking.** Several works (Power et al., 2022; Nanda et al., 2023; Lee et al., 2024) have observed the ability of Transformers to learn to perfectly perform small discrete algorithmic tasks, e.g. modular arithmetic.
- **Neural Algorithmic Reasoning.** Recent work (Rodionov & Prokhorenkova, 2023; 2024) investigates the ability of graph neural networks to learn fundamental algorithms like breadth-first search (Veličković et al., 2022).

Crucially, we note that these previous works focus on learning *discrete* algorithmic tasks, which Transformers excel at. As far as we know, we are the first to investigate whether Transformers are able to learn *numerical* algorithms, which rely on addressing key challenges with high-precision floating-point arithmetic.

**Precision and scientific ML.** The importance and difficulty of high-precision ML for scientific settings is well-established: although the scientific ML community has made exciting progress in recent years, numerical methods are still known to outperform existing ML methods in precision even on simple PDE benchmarks (McGreivy & Hakim, 2024). Despite this, we are aware of only a few works which directly focus on investigating high precision for ML. We highlight (Michaud et al., 2023; Wang & Lai, 2023), which focus on small MLPs for regression tasks and propose alternate training recipes.

As far as we are aware, we are the first to investigate and isolate effects of model architectures and optimizers on precision in a controlled setting: in-context least squares. We find that typical training recipes for sequence models (e.g. softmax attention, Adam, and standard LR schedulers) encounter surprising precision barriers when applied to numerical tasks.

# B EXPERIMENTAL SETUP

Here, we provide additional details about our experimental setup.

## B.1 MODEL ARCHITECTURE

We base our Transformer and BASECONV models off the GPT2 family (Radford et al., 2019). Unless otherwise specified, we use the following default settings for Transformers:

| Config | Setting |
|---|---|
| Embedding size | 64 |
| Number of layers | 12 |
| Number of heads | 1 |
| MLPs | True |
| MLP hidden size | $4\times$ embedding size |
| MLP activation | ReLU |
| LayerNorms | True |
| Input dim | 5 |
| Sequence length | 20 |

Table 1: Standard Transformer architecture details.

and the following settings for BASECONVs:

| Config | Setting |
|---|---|
| Embedding size | 64 |
| Number of layers | 3 |
| MLPs | False |
| LayerNorms | False |
| Input dim | 5 |
| Sequence length | 20 |

Table 2: BASECONV architecture details.

Finally, we describe the settings we use for our linear attention experiment (Figure 2):

| Config | Setting |
|---|---|
| Embedding size | 256 |
| Number of layers | 3 |
| Number of heads | 16 |
| MLPs | False |
| LayerNorms | False |
| Input dim | 5 |
| Sequence length | 20 |

Table 3: Linear attention architecture details.

## B.2 OPTIMIZER

We describe two sets of optimizer settings we use throughout this work.

The first, representative of standard training procedures, is inspired by prior in-context learning setups (Garg et al., 2022; Von Oswald et al., 2023).

The second, our training recipe, is for our high-precision experiments, where we find a more aggressive learning rate scheduler is essential. Note we use the adaptive learning rate scheduler and EMA described in Section 5.

| Config | Setting |
|---|---|
| Batch size | 256 |
| Optimizer | Adam |
| Learning rate | $10^{-3}$ |
| Scheduler | StepLR |
| Training iterations | $10^6$ |
| Step rate | $10^4$ |
| Decay rate | 0.9 |

Table 4: Standard optimizer settings.

| Config | Setting |
|---|---|
| Batch size | 1024 |
| Optimizer | Adam |
| Learning rate | $10^{-2}$ |
| Scheduler | AdaptiveLR |
| Training iterations | $2.5 \times 10^6$ |
| Step rate | $3 \times 10^3$ |
| Decay rate | 0.9 |
| EMA decay | 0.98 |
| EMA lambda | 2 |

Table 5: High-precision training recipe settings for BASECONV.

Finally, we describe the optimization settings we used for high-precision linear attention, which we found needed a slightly different learning rate scheduler.

| Config | Setting |
|---|---|
| Batch size | 1024 |
| Optimizer | Adam |
| Learning rate | $10^{-2}$ |
| Scheduler | AdaptiveLR |
| Training iterations | $2.5 \times 10^6$ |
| Step rate | $3 \times 10^3$ |
| Decay rate | 0.9 |
| EMA decay | 0.98 |
| EMA lambda | 2 |

Table 6: High-precision training recipe settings for linear attention.

### B.3 TASKS

Each of our in-context learning tasks can be viewed as a sequence-to-sequence map

$$\mathcal{M} : \mathbb{R}^{N_{in} \times D_{in}} \to \mathbb{R}^{N_{out} \times D_{out}}$$

In this subsection, we provide details about task implementations, specifying the input/output formats for each of the synthetic tasks and in-context least squares variants we implement.

#### B.3.1 IN-CONTEXT LEAST SQUARES.

We consider $\mathcal{M}_{LS} : \mathbb{R}^{N \times (D+1)} \to \mathbb{R}^D$, where as above the inputs are formatted as

$$\boldsymbol{u}_{in} := \begin{bmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N \\ b_1 & \dots & b_N \end{bmatrix}$$

and the expected output is

$$T_\theta(\boldsymbol{u}_{in})[\text{:-1, -1:}] := \boldsymbol{x}.$$

#### B.3.2 PRIMITIVES.

For each of the following linear algebra primitives, we increase the task size, setting $D = 20$ and $N = 40$.

- READ is defined as $\mathcal{M}_{Read} : \mathbb{R}^{N \times D} \to \mathbb{R}^{N \times D}$, where the inputs are formatted as

  $$\boldsymbol{u}_{in} \in \mathbb{R}^{N \times D} := [\boldsymbol{x}_1 \quad \dots \quad \boldsymbol{x}_N]$$

  and the expected outputs are $T_\theta(\boldsymbol{u}_{in}) \in \mathbb{R}^{N \times D}$ such that

  $$T_\theta(\boldsymbol{u}_{in})[k, :] := \begin{cases} \boldsymbol{u}_{in}[i, :] & k = j \\ \boldsymbol{u}_{in}[k, :] & k \neq j \end{cases}$$

  for task parameters $i \neq j \in [N]$.
- LINEAR is defined as $\mathcal{M}_{Linear} : \mathbb{R}^{N \times D} \to \mathbb{R}^{N \times 1}$, where the inputs are formatted as

  $$\boldsymbol{u}_{in} \in \mathbb{R}^{N \times D} := [\boldsymbol{x}_1 \quad \dots \quad \boldsymbol{x}_N]$$

  and the expected outputs are

  $$T_\theta(\boldsymbol{u}_{in}) := \begin{bmatrix} \boldsymbol{x}_1^T \boldsymbol{h} & \dots & \boldsymbol{x}_N^T \boldsymbol{h} \end{bmatrix}$$

  where $\boldsymbol{h} \in \mathbb{R}^D$ is a task parameter.
- MULTIPLY is defined as $\mathcal{M}_{Multiply} : \mathbb{R}^{N \times D} \to \mathbb{R}^{N \times D/2}$, where the inputs are formatted as

  $$\boldsymbol{u}_{in} \in \mathbb{R}^{N \times D} := [\boldsymbol{x}_1 \quad \dots \quad \boldsymbol{x}_N]$$

  and the expected outputs are

  $$T_\theta(\boldsymbol{u}_{in}) := \begin{bmatrix} (\boldsymbol{x}_1[:, : D/2] \odot \boldsymbol{x}_1[:, D/2 :] & \dots & \boldsymbol{x}_N[:, : D/2] \odot \boldsymbol{x}_N[:, D/2 :]) \end{bmatrix}.$$

#### B.3.3 EXPLICIT GRADIENT UPDATES.

In Section 5, we investigate a simple training setting, in which the model is explicitly trained to predict the gradient of the least squares loss. We proceed to define the task $\mathcal{M}_{gradient} : \mathbb{R}^{(N+1) \times (D^2+2D+1)} \to \mathbb{R}^D$.

The inputs are formatted as

$$\boldsymbol{u}_{in} := \begin{bmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{x}_0 \\ b_1 & \dots & b_N & 0 \end{bmatrix}.$$

The expected outputs are

$$T_\theta(\boldsymbol{u}_{in})[\text{-1:, :D}] := \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{x}_0).$$

#### B.3.4 $k$-TH GRADIENT DESCENT ITERATE.

Finally, toward end-to-end least squares, we investigate a series of increasingly end-to-end tasks in which the model is explicitly trained to predict the $k$-th gradient descent iterate. We proceed to define the task $\mathcal{M}^k_{iter} : \mathbb{R}^{(N+1)\times(D^2+2D+1)} \to \mathbb{R}^D$.

The inputs are formatted as

$$\boldsymbol{u}_{in} := \begin{bmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{x}_0 \\ b_1 & \dots & b_N & 0 \end{bmatrix}.$$

The expected outputs are

$$T_\theta(\boldsymbol{u}_{in})[\text{-1:}, \text{:D}] := \boldsymbol{x}_k.$$

### B.4 DATA GENERATION

At each training step, we produce a random training prompt $\boldsymbol{u}_{in}$ by sampling each variable randomly: from the isotropic Gaussian distribution $N(\boldsymbol{0}, \boldsymbol{I})$ for continuous-valued parameters, and from the uniform distribution for discrete parameters. Concretely:

- For the in-context linear regression tasks, input vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ are sampled from $N(\boldsymbol{0}^D, \boldsymbol{I}^D)$, and the unknown linear function is determined by $\boldsymbol{w}^*$, also drawn from $N(\boldsymbol{0}^D, \boldsymbol{I}^D)$.
- For the synthetic tasks READ, LINEAR, MULTIPLY (Section 3.3), *each column* of the inputs $\boldsymbol{u}_{in} \in \mathbb{R}^{N\times D}$ is sampled from the isotropic Gaussian distribution $N(\boldsymbol{0}^D, \boldsymbol{I}^D)$. The tasks READ and LINEAR require specifying additional parameters as follows:
  - For READ, at each iteration, $i \neq j \in [N]$ are sampled uniformly.
  - For LINEAR, at each iteration, the affine transformation $\boldsymbol{h}$ is sampled from $N(\boldsymbol{0}^D, 3\boldsymbol{I}^D)$.
- For the explicit gradient task and the $k$-th gradient descent iterate task, the random initialization $\boldsymbol{w}_0$ is also drawn from $N(\boldsymbol{0}^D, \boldsymbol{I}^D)$.

The model is trained to minimize mean squared error over the distribution of prompts.

# C ADDITIONAL EXPERIMENTAL RESULTS

## C.1 ABLATIONS: LINEAR ALGEBRA PRIMITIVES

In Figure 6, we train Transformers and BASECONVs, *with* MLPs, with and without LayerNorms (LN), on the READ, LINEAR, and MULTIPLY primitives from Section B.3.2. We vary the model depth $L \in \{1, 2, 4, 8\}$ and investigate how precision scales with number of layers. In these experiments, we use a standard exponentially decaying LR schedule for Adam.

We show that Transformers and BASECONVs both achieve high precision ($< O(10^{-9})$) on the READ and LINEAR tasks. However, the Transformers struggle to implement MULTIPLY to high precision, and performance scales poorly with model depth. We observe that BASECONV without LayerNorm generally performs the best across all three primitives, consistently outperforming BASECONV with LayerNorm by 2-4 orders of magnitude.

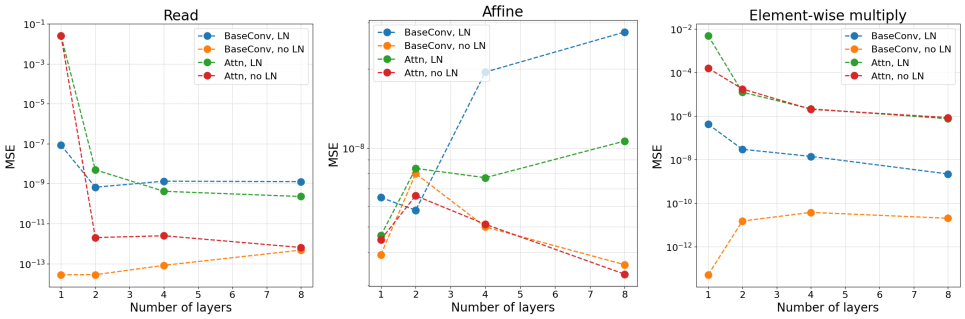


Figure 6: Attention vs. BASECONV, with and without LayerNorms, on synthetic tasks. Precision consistently scales better with depth for BASECONV models than for Transformers. While both models solve READ and LINEAR tasks to at least $10^{-8}$ MSE, the precision of Transformers scales poorly for the MULTIPLY task.

Focusing on 2-layer Transformers and the MULTIPLY task, we additionally find that precision scales poorly with multiple scaling axes, including hidden dimension, number of heads, and MLP upscaling factor (Figure 7).

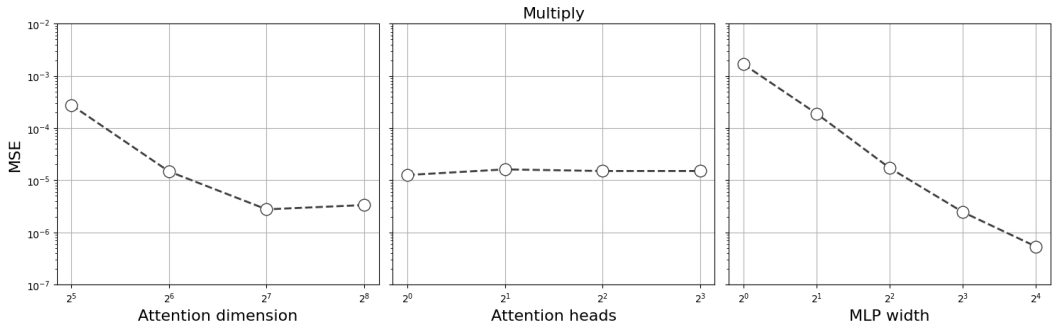


Figure 7: Precision of (2-layer) Transformers on MULTIPLY task scales poorly with attention dimension (left), number of heads (middle), and MLP width (right, where MLP hidden dimension = width $\times$ attention dimension).

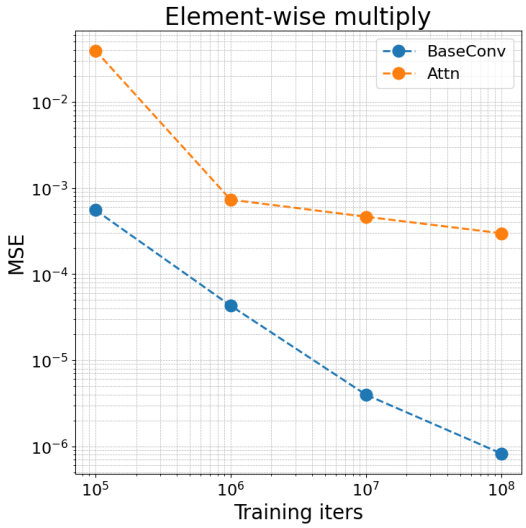


Figure 8: Scaling number of training iterations for 1-layer Transformer vs. BASECONV on the MULTIPLY task. Both models improve precision by 2-3 orders of magnitude as training duration increases by 3 orders of magnitude.

Finally, we investigate the effect of training duration on precision. In Figure 8, we train 1-layer Transformers and BASECONVs, with MLPs and LayerNorms, on the MULTIPLY primitive and vary the number of iterations for which the model is trained. Recall that since new data is sampled at each iteration, we also effectively scale the dataset size proportionally. To keep the learning rates consistent across runs, we scale back the scheduler step size accordingly:

$$num\_iters \in \{10^5, 10^6, 10^7, 10^8\}$$
$$step\_size \in \{10^3, 10^4, 10^5, 10^6\}$$

We observe a power law, particularly clearly for BASECONV, as we scale from $10^5$ to $10^8$ iterations. Both models achieve a 2-3 order of magnitude improvement in precision, but this requires also increasing training duration by 3 orders of magnitude.

### C.2 ABLATIONS: HIGH-PRECISION OPTIMIZATION

In Figure 9, we try directly training on the end-to-end least squares task, simply replacing softmax attention with BASECONV in the standard Transformer architecture. We find we are unable to reach high precision using this training procedure.

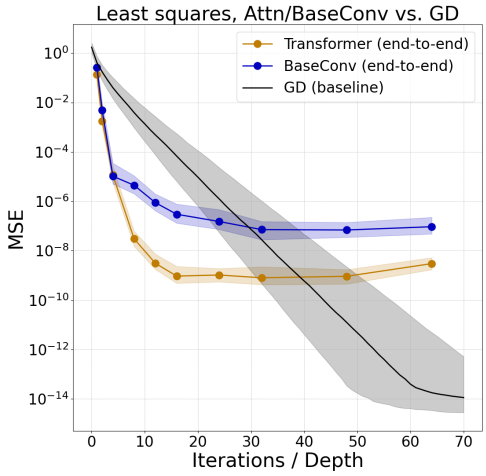


Figure 9: Replacing softmax attention with BASECONV in the standard Transformer architecture and training end-to-end on least squares is not enough to achieve high-precision solutions. BASECONV models trained end-to-end perform as badly as Transformers at small scale, and our largest models perform $100\times$ *worse* than parameter-matched Transformers.

In Figure 10, we ablate the effects of constant and exponentially decaying LR schedulers with Adam (cutting off training after $10^6$ iterations). We find that neither are able to efficiently train to machine precision on the explicit gradients task. For exponentially decaying LR schedule, we find that the LR steprate is a crucial parameter: on the explicit gradient task, a difference of $10,000\times$ between precision saturation thresholds using $1 \times 10^3$ vs $3 \times 10^3$ for example.

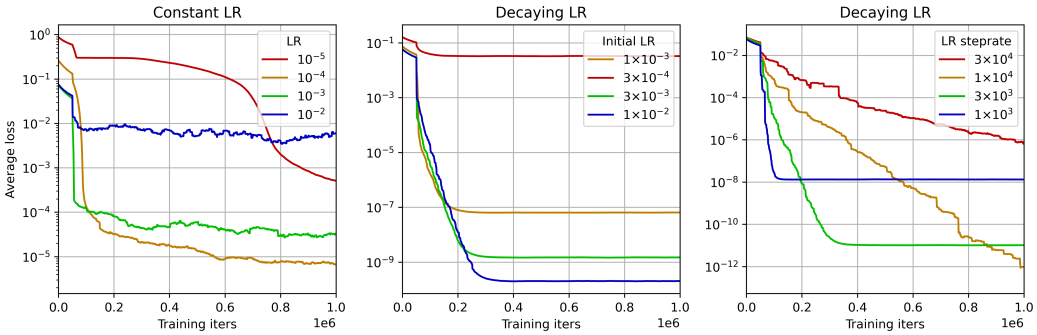


Figure 10: Training with Adam on the explicit gradient task, we ablate LR for constant scheduler (left), initial LR (middle) and LR steprate (right) for decaying scheduler.

In Figure 11, we ablate the effect of applying an EMA over the update vectors from the Adam optimizer. Empirically, we find that this boosts the final MSE by as much as $100,000\times$ on the explicit gradient task.

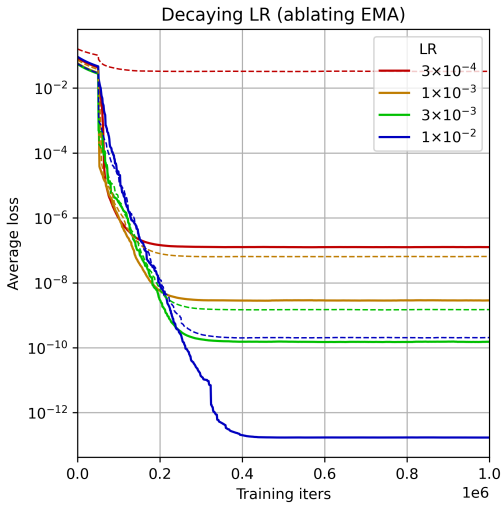


Figure 11: Training on the explicit gradient task, applying EMA over Adam's update vectors consistently boosts final MSE, by up to 5 orders of magnitude.

In Figure 12, we ablate the effect of restoring the MLPs and LayerNorms to BASECONV models. Surprisingly, we find that even these architectural components worsen the model's precision: on the explicit gradient task, by a factor of up to $1,000,000\times$ MSE. We note that due to training instability with the BASECONV+MLP model, we used a less aggressive LR schedule with initial LR $10^{-3}$ and LR steprate $10^4$.

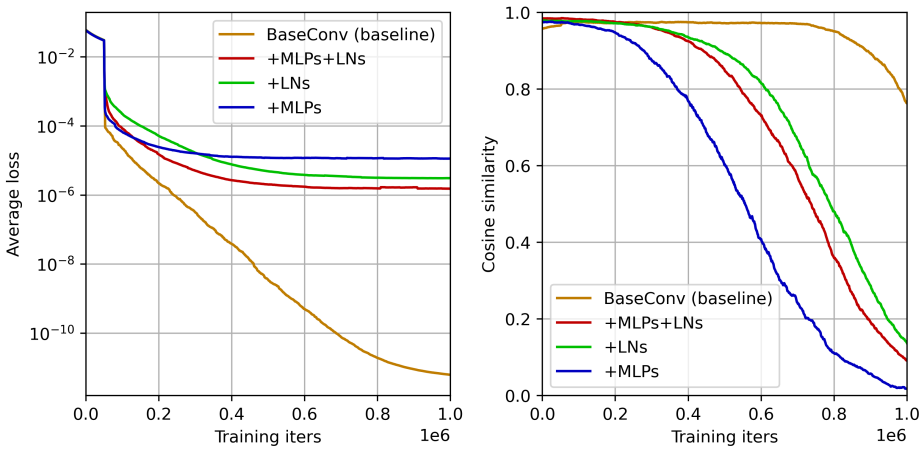


Figure 12: Training on the explicit gradient task, adding MLPs and LNs consistently bottlenecks precision: here, by up to 6 orders of magnitude.

In Figure 13, we evaluate 3-layer BASECONV and linear attention models trained on the explicit gradient task. As in Section 5.2, we apply them iteratively until convergence. We then evaluate on out-of-distribution regression targets, as in Section 3.2.

We surprisingly find that linear attention demonstrates poor numerical generality, despite training to near machine precision on the training distribution. Beyond $\sigma = 4$, the iterations of linear attention diverge.

This result suggests that although different polynomial architectures may equally be able to *express* algorithms, they may *learn* solutions that exhibit vastly different numerical properties.

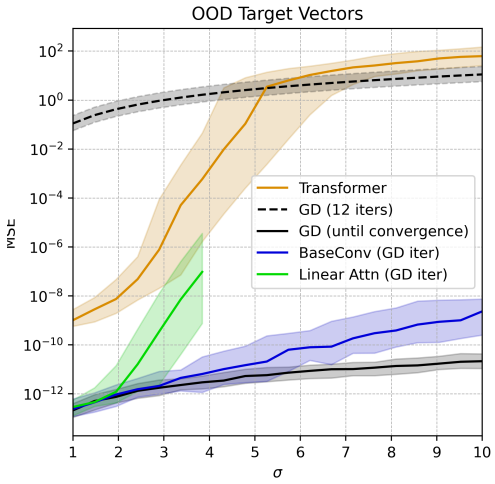


Figure 13: While BASECONV demonstrates improved numerical generality compared to end-to-end trained Transformers, the generalization gap for linear attention is as bad as the Transformer.

### C.3 $k$-TH ITERATE GD

In this section, we investigate how well our proposed techniques can learn the $k$-th GD iterate tasks as defined in Section 5:

$$\{(\boldsymbol{a}_1, b_1), \ldots, (\boldsymbol{a}_N, b_N), \boldsymbol{x}_0\} \to \boldsymbol{x}_k, \text{ where } \boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \eta \nabla \mathcal{L}(\boldsymbol{x}_i),\ i \in [k-1]. \tag{13}$$

Recall that $k = 1$ is equivalent to the explicit gradient task, while taking $k \to \infty$ is equivalent to the standard in-context least squares task. Here, we are interested in understanding how well our techniques extend to larger $k$, towards learning end-to-end least squares. See Appendix B for a more detailed description of the training setup.

Our theoretical results in Section 4 imply that a $k+2$-layer BASECONV is expressive enough to solve the $k$-th iterate task to machine precision. Thus we train $k+2$-layer BASECONV models on the $k$-th iterate task for $k \geq 1$ using our training recipe.

| $k$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| MSE | $5.0 \times 10^{-13}$ | $2.5 \times 10^{-11}$ | $2.5 \times 10^{-11}$ | $3.1 \times 10^{-10}$ |

Table 7: We can learn up to 4 iterations of GD at once with our current training techniques. Model stability becomes a bottleneck with harder tasks.

Training on the $k$-iter GD task, we find our training recipe scales to $k = 4$ before training instability occurs. Adding LayerNorms, we are able to train deeper models, but we find MSE worsens by at least $1,000\times$ for small $k$: see Figure 14.

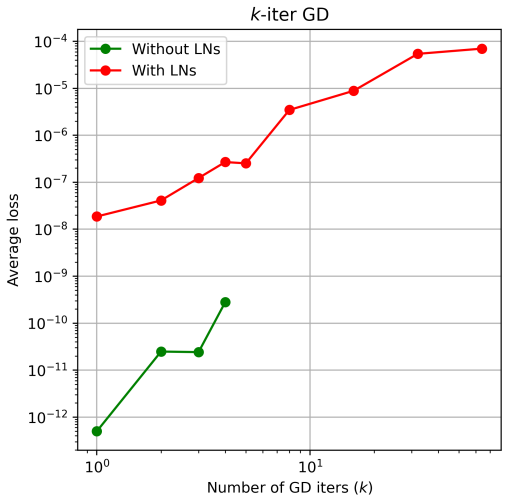


Figure 14: BASECONV with LayerNorms are able to stably scale to deeper models, but LayerNorms present a precision bottleneck: even on small $k$, MSE degrades by over $1,000\times$.

### C.4 IN-CONTEXT ODE SOLVING

In this section, we demonstrate the generality of our insights on the more practical setting of in-context ODE solving. We note that solving differential equations in-context with Transformers is a framework that has been explored in recent papers (Yang et al., 2023a; Herde et al., 2024; Liu et al., 2023a), and thus represents a natural first step towards extending our techniques to realistic scientific ML problems.

**Experimental setup.** We follow the setup from Liu et al. (2023a):

- We train on a distribution of 1D ODEs over $t \in [-1, 1]$, defined by
$$u'(t) = \alpha_1 c(t) + \alpha_2 u(t) + \alpha_3. \tag{14}$$
For each operator, we provide 25 in-context examples of forcing functions, initial conditions, and their corresponding solution values at a fixed time $t_{query} \in [-1, 1]$. We then give the model a query forcing function and initial condition, and the goal is to predict the corresponding solution at $t_{query}$.
- We sample our parameters $\alpha_1 \sim \text{Unif}([0.5, 1.5])$, $\alpha_2 \sim \text{Unif}([-1, 1])$, $\alpha_3 \sim \text{Unif}([-1, 1])$.
- Initial conditions are sampled from $u(0) \sim \text{Unif}([-1, 1])$.
- Forcing functions $c(t)$ are sampled from a Gaussian process with RBF kernel $K(x, x') = \exp\left(-\frac{(x-x')^2}{2\ell^2}\right)$, with length-scale parameter $\ell = 1$. We sample each forcing function on 21 equispaced points over $[-1, 1]$.
- ODEs are solve pseudospectrally on $N = 41$ nodes: we find this is sufficient for machine-precision solutions with FLOAT32 datatype.

We find that our observations from least squares transfer to the setting of in-context ODEs:

**Transformers struggle to learn precise solutions.** We find that a 12-layer, 9M parameter Transformer model only achieves $\approx 10^{-4}$ MSE, almost $10^{10}\times$ worse than the threshold FLOAT32 machine epsilon implies. Furthermore, as with least squares, we observe precision saturation with model size. In Figure 15, we find that scaling the depth of the model by up to $2\times$ does not improve precision. We further note that precision saturations already seems to occur with 4-layer Transformers. We hypothesize that the depth at which precision saturation begins is dependent on the task difficulty.

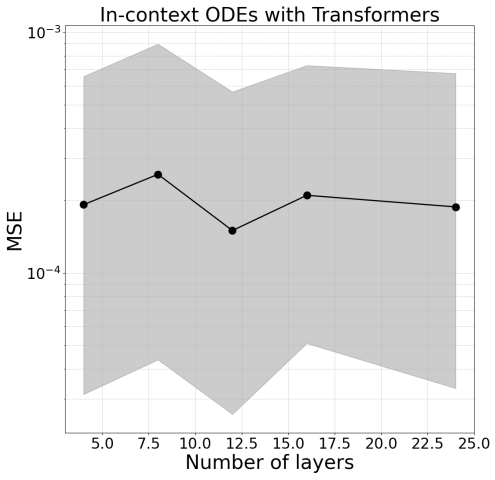


Figure 15: Transformers fail to learn precise algorithms for solving ODEs in-context. As with least squares, precision saturates with deeper models: in our experiments, we observe no significant performance boost between 4-layer and 24-layer Transformers.

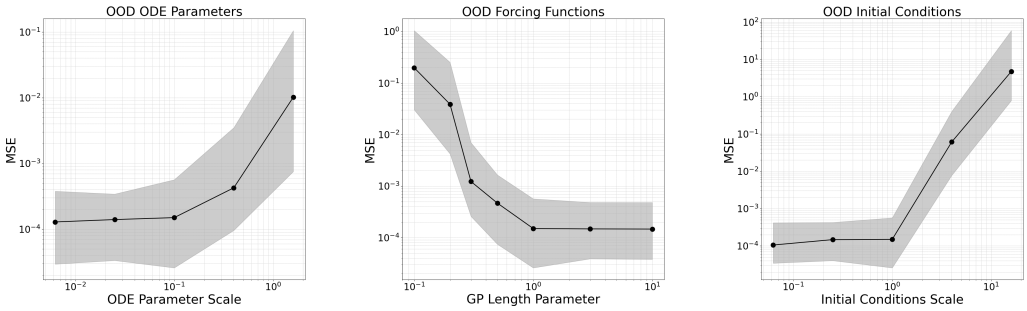


Figure 16: Transformers fail to learn numerically general solutions: performance is brittle to out-of-distribution ODE parameters (left), forcing function smoothness (middle), and initial condition distribution (right).

**Transformers exhibit brittle generalization.** We observe that Transformers are not robust to changes to the distributions of ODE parameters, forcing functions, and initial conditions. We describe our experimental setup below, mirroring Liu et al. (2023a):

- Out-of-distribution ODE parameters. We parameterize out-of-distribution ODEs via a scale parameter $\sigma_{op}$, where $\alpha_1 \sim \text{Unif}([1-\frac{1}{2}\sigma_{op}, 1+\frac{1}{2}\sigma_{op}])$ and $\alpha_2, \alpha_3 \sim \text{Unif}([-\sigma_{op}, \sigma_{op}])$. As we increase $\sigma_{op}$, we sample from a wider distribution of ODE solution operators, including those with larger operator norms and worse-conditioned design matrices.
- Out-of-distribution forcing functions. We vary $\ell$, the length parameter of the Gaussian process from which we sample our forcing functions, which effectively controls their smoothness.
- Out-of-distribution initial conditions. We sample out-of-distribution initial conditions as $u(0) \sim \text{Unif}([-\sigma_{IC}, \sigma_{IC}])$. As we vary $\sigma$, we widen the distribution of the solution values at $t = 0$, which increases the overall magnitudes of the solutions.

We note that in all out-of-distribution experiments, the Transformer's MSE explodes to near $O(1)$: refer to Figure 16.

**Our proposed techniques obtain precise and general solutions.** Liu et al. (2023a) shows that in-context ODEs can be reduced to solving least squares problems. Thus, we train a 3-layer BASECONV architecture on the explicit gradient task for the equivalent least squares problem, and apply our model iteratively, as in Section 5. We compare the performance of our iterative model with end-to-end Transformers, least squares solvers, and standard gradient descent applied to the equivalent least squares problem.

We note that our ODEs reduce to least squares problems that are ill-conditioned. In this set of experiments, we find the condition numbers of our design matrices are $O(10^8)$. Since the theoretical convergence rate of gradient descent on least squares depends inversely on the condition number (Boyd & Vandenberghe, 2004), we expect our iterative models and standard gradient descent will require orders of magnitude more iterations than in the least squares problems of Section 5. As such, we limit the number of iterations for our BASECONV model and standard gradient descent to $10,000$. Nonetheless, we find that our BASECONV model learns to high enough precision that we are able to maintain the stability of the iterative algorithm for up to $O(10^5)$ steps. In our experiments, we iteratively apply our BASECONV model until convergence to a fixed point and report final MSEs.

We find that BASECONV learns a precise and general algorithm for in-context ODEs:

- **Precision.** In Figure 17, we show that our BASECONV model, applied iteratively, converges to $\approx 10^{-10}$ MSE, $\approx 1,000,000\times$ higher precision than our best Transformers.
- **Generality.** In Figure 18, we find that our BASECONV model exhibits more robust generalization than the Transformer model: in all the out-of-distribution settings we test, our BASECONV model achieves higher precision than Transformers in-distribution. Like above,

we evaluate on out-of-distribution ODE parameters, forcing functions, and initial conditions. In particular, we note that the performance of our BASECONV model almost exactly matches proper gradient descent, even in out-of-distribution settings. Additionally, we find the generalization behavior of our BASECONV model matches the generalization of a proper least squares solver with preconditioning, except for out-of-distribution initial conditions, where we note that the iterative procedure suffers from slow convergence and times out at $10,000$ iterations.

We believe these preliminary results show the promise of our techniques towards learning numerical algorithms for more complex tasks, such as solving PDEs, directly from data.

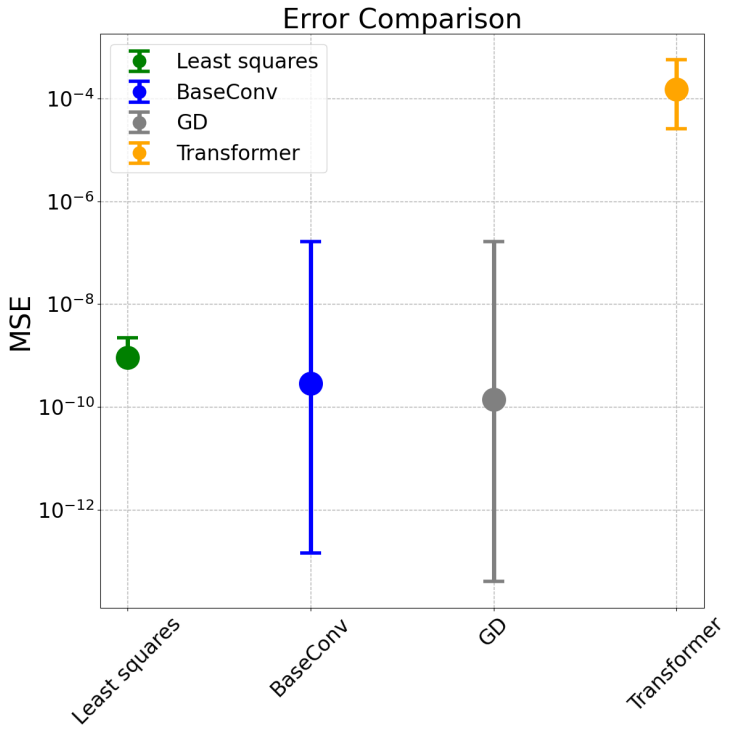


Figure 17: In-distribution error comparison between Transformer, BASECONV, gradient descent, and least squares.

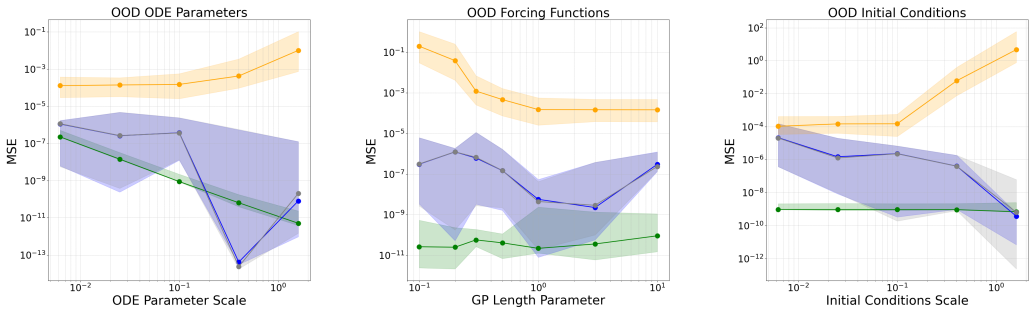


Figure 18: Out-of-distribution error comparison between Transformer (orange), BASECONV (blue), gradient descent (gray), and least squares (green): we evaluate out-of-distribution ODE parameters (left), forcing function smoothness (middle), and initial condition distribution (right). BASECONV learns a numerically general algorithm that closely matches proper gradient descent and least squares.

# D THEORETICAL RESULTS

This section is organized as follows:

- We detail notation and definitions in Appendix D.1, Appendix D.6.
- In Appendix D.2, we include theoretical results regarding the primitives from Section 3.3: expressivity results with BASECONV and attention, and iterative algorithms as compositions of primitives.
- In Appendix D.3, we discuss upper and lower bounds for implementing gradient descent on least squares using BASECONV, supplementing Section 4.1.
- In Appendix D.4, we provide theoretical details regarding BASECONV and polynomials.
- Finally, in Appendix D.7, we provide technical details about the claims from Section 4.1 that BASECONV can perfectly recover Square and Linear.

## D.1 NOTATION

We heavily borrow notation from Appendix H of Arora et al. (2023), which we recollect below. We denote the all 1 row vector of size $k$, given by $[1 \quad 1 \quad \dots \quad 1 \quad 1]$, and the all 0 row vector of size $k$, given by $[0 \quad 0 \quad \dots \quad 0 \quad 0]$, as $\mathbf{1}^k$ and $\mathbf{0}^k$, respectively. We also construe the standard basis vector $\mathbf{e}_i$ as a column vector in this appendix, and adhere to the following matrix indexing convention: $\mathbf{M}[i,j]$ is the entry in the $i$th row and the $j$th column, $\mathbf{M}[i,:] \in \mathbb{F}^{1\times n}$ denotes the $i$th row, and $\mathbf{M}[:,j] \in \mathbb{F}^{m\times 1}$ denotes the $j$th column of $\mathbf{M} \in \mathbb{F}^{m\times n}$, where $\mathbb{F}$ is a field (the reader can assume that $\mathbb{F}$ is the field of real numbers i.e. $\mathbb{F} = \mathbb{R}$). We then use $\mathbf{1}^{m\times n}, \mathbf{0}^{m\times n} \in \mathbb{F}^{m\times n}$ to denote the matrix of all 1s and 0s, respectively. We note that some notation differs from those used in earlier sections.

Next, we denote the *Hadamard product* of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ as $\mathbf{u} \odot \mathbf{v}$; the operation can be extended to matrices by applying the Hadamard product column-wise across the matrices. This is commonly referred to as *(element-wise) gating*. For vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$, we also denote their *linear (or acyclic) convolution* as $\mathbf{u} * \mathbf{v}$ and *cyclic convolution* as $\mathbf{u} \circledast \mathbf{v}$.

**Polynomial Notation.** Since convolution is equivalent to operations on polynomials, it is convenient to use them to discuss the inputs and outputs of gated convolution models. Let us define maps $\text{poly} : \mathbb{F}^n \to \mathbb{F}[X]/(X^n)$ such that

$$\text{poly}(\boldsymbol{u}) = \sum_{i=0}^{n-1} \boldsymbol{u}[i] X^i.$$

This allows us to map between vectors and polynomial. Accordingly, we also define $\text{coeff} : \mathbb{F}[X]/(X^{n+1}) \to \mathbb{F}^n$ as the map converting polynomials back to vectors: $\text{coeff}(\boldsymbol{u}(X)) = \boldsymbol{u}$ with $\boldsymbol{u}[i]$ defined as the coefficient in $\boldsymbol{u}(X)$ at degree $i$.

These operations allow us to interpret the convolution of vectors in terms of polynomial multiplication (Heideman & Burrus, 1988). More specifically, we have

$$\boldsymbol{u} * \boldsymbol{v} = \text{coeff}\left(\boldsymbol{u}(X) \cdot \boldsymbol{v}(X) \mod X^n\right)$$

The following notation for a polynomial will be used in this section:

**Definition D.1.** A polynomial $P(X)$ with degree $d$ and some coefficients $\mathbf{c} \in \mathbb{R}^{d+1}$ is defined as,

$$P(X) = \sum_{i=0}^{d} c_i X^i.$$

Further, the degree of $P(X)$ will be denoted as $\deg(P)$.

**Function Approximation.** In this part, we collect notation and known results about function approximation. We will reference some definitions from Pleśniak (2009); Petersdorff (2015); Smoothness (2006).

The following notation is to denote the $k$th derivative of a function:

**Definition D.2.** For some function $f : \mathbb{R} \to \mathbb{R}$, $f^{(k)} := \frac{d^k}{dx^k} f(x)$ is the $k$th derivative of $f$.

Define a set of univariate functions with a notion of continuity:

**Definition D.3.** We denote $C^k[a, b]$ for $k = 1, 2, \ldots$ the space of univariate functions $f : [a, b] \to \mathbb{R}$, which have derivatives $f^{(1)}, \ldots, f^{(k)}$ that are continuous on the closed interval $[a, b]$.

Next we define a set of multivariate functions with a notion of continuity:

**Definition D.4.** A function $f : [a, b]^n \to \mathbb{R}$ is in $C^k[a, b]^n$ for $k = 1, 2, \ldots$ if all partial derivatives

$$\frac{\partial^\alpha}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \cdots \partial x_n^{\alpha_n}} f(y_1, y_2, \ldots, y_n)$$

exist and are continuous, for every $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{Z}_{\geq 0}$, such that $\alpha_1 + \alpha_2 + \cdots + \alpha_n \leq k$ and every $(y_1, \ldots y_n) \in [a, b]^n$.

We use the following notation for the set of all univariate polynomials:

**Definition D.5.** For any integer $d \geq 0$, we define

$$\mathcal{P}_d(X) = \{c_0 + c_1 X + \cdots + c_d X^d | c_k \in \mathbb{R}\}.$$

In other words, $P_d(X)$ is the space of univariate polynomials of degree less or equal to $d$.

We use the following notation for multivariate polynomials:

**Definition D.6.** For any integers $n, d \geq 0$ , we define

$$\mathcal{P}_d^n(X_1, \ldots, X_n) = \left\{ \sum_{\boldsymbol{\alpha}=(\alpha_1,\ldots,\alpha_n) \in \mathbb{Z}_{\geq 0}^n} c_\alpha X_1^{\alpha_1} X_2^{\alpha_2} \cdots X_n^{\alpha_n} \middle| c_\alpha \in \mathbb{R}, \sum_{i=0}^{n} \alpha_i \leq d \right\}.$$

Then $\mathcal{P}_d^n(X_1, \ldots X_n)$ is the space of $n$-variate polynomials of degree less or equal to $d$.

The following notation is for considering the pointwise absolute value of a matrix:

**Definition D.7.** For $\boldsymbol{M} \in \mathbb{R}^{N \times D}$ define,

$$\|\boldsymbol{M}\|_\infty = \max_{\substack{0 \leq i < N \\ 0 \leq j < D}} |M[i, j]| .$$

Now lets define the corresponding $\infty-$norm for functions:

**Definition D.8.** For $g : [-1, 1]^{N \times D} \to \mathbb{R}^{N \times D}$, define

$$\|g\|_\infty = \max_{\mathbf{x} \in [-1,1]^{N \times D}} |g(\mathbf{x})| .$$

We will use the following version of Jackson's theorem for univariate inputs:

**Theorem D.9** (D. Jackson (1930) Jackson's Theorem for $C^k[-1, 1]$.)**.** *Let $d, k$ be integers with $d + 1 \geq k \geq 0$ and $f \in C^k[-1, 1]$. Then*

$$\inf_{P \in \mathcal{P}_d} \|f - P\|_\infty \leq \left(\frac{\pi}{2}\right)^k \frac{1}{(d+1)d \cdots (d-k+2)} \left\| f^{(k)} \right\|_\infty. \tag{15}$$

We will use the following version of Jackson's theorem for multivariate inputs:

**Theorem D.10** (Pleśniak (2009) Jackson's Theorem for $C^k[-1, 1]^n$)**.** *Let $d, k$ be integers with $d + 1 \geq k \geq 0$ and $f \in C^k[-1, 1]^n$. Then*

$$\inf_{P \in \mathcal{P}_d^n} \|f - P\|_\infty \leq \frac{c_k}{d^k} \sum_{j=1}^{n} \left\| \frac{\partial^{k+1}}{\partial x_j^{k+1}} f(\mathbf{x}) \right\|_\infty \tag{16}$$

*where $c_k$ is a positive constant.*

We will use the following definition of univariate smooth functions:

**Definition D.11.** We call a $k$ times differentiable function $f : [-1, 1] \to \mathbb{R}$ to be $(k, L)$-smooth if $\left\|f^{(k)}\right\|_\infty \leq L$.

Next, we observe that given a univariate smooth function, there's a univariate bounded degree polynomial that approximates it to some error, $\epsilon$:

**Corollary D.12.** *For some $(k, L)$-smooth univariate function $f$ (as in Definition D.11), then there exists a polynomial $P_f(x)$ with*

$$\deg(P_f) \leq O\left(\sqrt[k]{\frac{L}{\epsilon}}\right) + k$$

*such that for all $x \in [-1, 1]$*

$$|f(x) - P_f(x)| \leq \epsilon.$$

*Proof.* We will be a bit more specific on an upper bound of $\deg(P_f)$. We pick:

$$\deg(P_f) = \left\lceil \frac{\pi}{2}\left(\frac{L}{\epsilon}\right)^{\frac{1}{k}} + k \right\rceil. \tag{17}$$

Let $d = \deg(P_f)$ where $P_f$ is the polynomial that achieves the left hand side of Equation (15). Then we have error at most

$$\left(\frac{\pi}{2}\right)^k \frac{1}{(d+1)d\cdots(d-k+2)}\left\|f^{(k)}\right\|_\infty.$$

Using the definition of a $(k, L)$-smooth univariate function in Definition D.11 we get the error at most

$$\left(\frac{\pi}{2}\right)^k \frac{L}{(d+1)d\cdots(d-k+2)} \leq \left(\frac{\pi}{2}\right)^k \frac{L}{(d-k)^k}$$

where the inequality follows since each $d+1, d, \ldots, d-k+2 \geq (d-k)$.

Plugging in Equation (17) for $d$ we get the error is at most:

$$\left(\frac{\pi}{2}\right)^k \frac{L}{\left(\frac{\pi}{2}\right)^k \left(\sqrt[k]{\frac{L}{\epsilon}}\right)^k} = \epsilon,$$

as desired. □

We will use the following definition of multivariate smooth functions that map to a single value:

**Definition D.13.** We call a $k$ times differentiable $f : [-1, 1]^n \to \mathbb{R}$ to be $(k, L)$-smooth if $\left\|\frac{\partial^k}{\partial x_m^k} f(\mathbf{x})\right\|_\infty \leq L$ for all $1 \leq m \leq n$.

Now we show the corresponding observation for multivariate functions and polynomials:

**Corollary D.14.** *Let $\deg(P_f) = d$. For some $(k, L)$-smooth multivariate function $f$ (as in Definition D.13), then there exists a polynomial $P_f(\mathbf{x})$ with*

$$\deg(P_f) \leq O_k\left(\sqrt[k]{\frac{n\,L}{\epsilon}}\right)$$

*such that for all $\mathbf{x} \in [-1, 1]^n$*

$$|f(\mathbf{x}) - P_f(\mathbf{x})| \leq \epsilon.$$

*Proof.* Let $P_f$ be the polynomial we get from the left hand side of Equation (16). We want to upper bound the error as

$$\frac{c_k}{d^k}\sum_{j=1}^{n}\left\|\frac{\partial^{k+1}}{\partial x_j^{k+1}} f(\mathbf{x})\right\|_\infty \leq \epsilon,$$

which follows if

$$\frac{c_k}{d^k}\sum_{j=1}^{n} L \leq \epsilon$$

since $f$ is $(k, L)$-smooth. The above is the same as

$$\frac{c_k n L}{d^k} \leq \epsilon,$$

or equivalently

$$\sqrt[k]{\frac{c_k n L}{\epsilon}} \leq d.$$

Picking $d = \left\lceil \sqrt[k]{\frac{c_k n L}{\epsilon}} \right\rceil$ suffices. □

**Arithmetic Circuit Notation.** We briefly recall arithmetic circuits Peter Bürgisser and Michael Clausen and M. Amin Shokrollah (1997). An *arithmetic circuit* $\mathcal{C}$ with variables $X \triangleq \{x_1, x_2, \ldots, x_n\}$ over a field $\mathbb{F}$ is interpreted as a directed acyclic graph, where the input nodes are labelled by either the variables from $X$ or constants from $\mathbb{F}$ and the internal nodes are labelled by $+$ or $\times$ with the output being the polynomial computed at the output node.

We shall also refer to the *size*[1] of the circuit $\mathcal{C}$ as the number of wires (or edges in $\mathcal{C}$), the *depth* of the circuit as the length of the longest path between an input node and the output node, and the *width* of the circuit as the number of wires that will be intersected by a horizontal 'cut' through the circuit. Moreover, the *degree* of a circuit is defined as the degree of the polynomial computed by the circuit. We summarize this with the following definition:

**Definition D.15.** An arithmetic circuit $\mathcal{C}$ is an $(n, s, \Delta, w)$-*circuit* if $\mathcal{C}$ is an $n$-variate arithmetic circuit of size $s$, depth at most $\Delta$, and width $w$.

**BASECONV Architecture.** In the following definitions we formally define the BASECONV model Arora et al. (2023). To formally define BASECONV, we will need the Kaleidoscope hierarchy Dao et al. (2020) as well.

To start, we define butterfly factors:

**Definition D.16.** A **butterfly factor** of size $k \geq 2$ (denoted as $\overline{\mathbf{B}}_k$) is a matrix of the form $\overline{\mathbf{B}}_k = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$ where each $\mathbf{D}_i$ is a $\frac{k}{2} \times \frac{k}{2}$ diagonal matrix. We restrict $k$ to be a power of 2.

The following definition is for a butterfly factor matrix, which is made up of the above butterfly factors:

**Definition D.17.** A **butterfly factor matrix** of size $n$ with block size $k$ (denoted as $\overline{\mathbf{B}}_k^{(n)}$) is a block diagonal matrix of $\frac{n}{k}$ (possibly different) butterfly factors of size $k$:

$$\overline{\mathbf{B}}_k^{(n)} = \operatorname{diag}\left(\left[\overline{\mathbf{B}}_k\right]_1, \left[\overline{\mathbf{B}}_k\right]_2, \ldots, \left[\overline{\mathbf{B}}_k\right]_{\frac{n}{k}}\right)$$

Now lets define a butterfly matrix:

**Definition D.18.** A **butterfly matrix** of size $n$ (denoted as $\overline{\mathbf{B}}^{(n)}$) is a matrix that can be expressed as a product of butterfly factor matrices: $\overline{\mathbf{B}}^{(n)} = \overline{\mathbf{B}}_n^{(n)} \overline{\mathbf{B}}_{\frac{n}{2}}^{(n)} \ldots \overline{\mathbf{B}}_2^{(n)}$. Equivalently, we may define $\overline{\mathbf{B}}^{(n)}$ recursively as a matrix that can be expressed in the following form:

$$\overline{\mathbf{B}}^{(n)} = \overline{\mathbf{B}}_n^{(n)} \begin{bmatrix} [\overline{\mathbf{B}}^{(\frac{n}{2})}]_1 & 0 \\ 0 & [\overline{\mathbf{B}}^{(\frac{n}{2})}]_2 \end{bmatrix}$$

(Note that $[\overline{\mathbf{B}}^{(\frac{n}{2})}]_1$ and $[\overline{\mathbf{B}}^{(\frac{n}{2})}]_2$ may be different.)

[1]Note that if all the gates of an arithmetic circuit have bounded arity then the number of wires and gates are asymptotically the same but in this appendix we will consider gates with unbounded arity.

Using these butterfly matrices, lets define the Kaleidoscope Hierarchy:

**Definition D.19** (The Kaleidoscope Hierarchy (Dao et al., 2020))**.**

- Define $\mathcal{B}$ as the set of all matrices that can be expressed in the form $\overline{\mathbf{B}}^{(n)}$ (for some $n$).
- Define $(\mathcal{B}\mathcal{B}^*)$ as the set of matrices $\mathbf{M}$ of the form $\mathbf{M} = \mathbf{M_1}\mathbf{M}_2^*$ for some $\mathbf{M_1}, \mathbf{M}_2 \in \mathcal{B}$.
- Define $(\mathcal{B}\mathcal{B}^*)^w$ as the set of matrices $\mathbf{M}$ that can be expressed as $\mathbf{M} = \mathbf{M}_w \dots \mathbf{M}_2\mathbf{M}_1$, with each $\mathbf{M}_i \in (\mathcal{B}\mathcal{B}^*)$ $(1 \le i \le w)$. (The notation $w$ represents width.)
- Define $(\mathcal{B}\mathcal{B}^*)_e^w$ as the set of $n \times n$ matrices $\mathbf{M}$ that can be expressed as $\mathbf{M} = \mathbf{S}\mathbf{E}\mathbf{S}^\top$ for some $en \times en$ matrix $\mathbf{E} \in (\mathcal{B}\mathcal{B}^*)^w$, where $\mathbf{S} \in \mathbb{F}^{n \times en} = [\mathbf{I}_n \quad 0 \quad \dots \quad 0]]$ (i.e. $\mathbf{M}$ is the upper-left corner of $\mathbf{E}$). (The notation $e$ represents expansion relative to $n$.)

Here we now formally define a BASECONV layer:

**Definition D.20** (BASECONV (Arora et al., 2023))**.** Given an input sequence $\mathbf{u} \in \mathbb{R}^{N \times D}$, where $N$ is the sequence length and $D$ is the model dimension, a learned weight matrix $\boldsymbol{W} \in \mathbb{R}^{D \times D}$ and biases $\boldsymbol{B}_1, \boldsymbol{B}_2 \in \mathbb{R}^{N \times D}$ and a matrix of convolution filters $\boldsymbol{H} \in \mathbb{R}^{N \times D}$, a BASECONV layer computes the following:

$$\boldsymbol{y}^{\text{BASECONV}} := (\mathbf{u}\boldsymbol{W} + \boldsymbol{B}_1) \odot (\boldsymbol{H} * \mathbf{u} + \boldsymbol{B}_2) \in \mathbb{R}^{N \times D}, \tag{18}$$

where the $j$th column of $\boldsymbol{H} * \mathbf{u} \in \mathbb{R}^{N \times D}$ is defined as $\boldsymbol{H}[:, j] * \mathbf{u}[:, j]$.

The corresponding pseudocode for a BASECONV layer is as follows:

**Algorithm 1** BASECONV$(\mathbf{u}, \boldsymbol{W}, \boldsymbol{B}_1, \boldsymbol{H}, \boldsymbol{B}_2)$

**Require:** Input sequence $\mathbf{u} \in \mathbb{R}^{N \times D}$, linear map $\boldsymbol{W} \in \mathbb{R}^{D \times D}$, convolution filter $\boldsymbol{H} \in \mathbb{R}^{N \times D}$, and bias matrices $\boldsymbol{B}_1, \boldsymbol{B}_2 \in \mathbb{R}^{N \times D}$.

1: In parallel for $0 \le n < N$ : $\boldsymbol{x}[n, :] = \mathbf{u}[n, :] \cdot \boldsymbol{W}$
2: In parallel for $0 \le t < D$ : $\boldsymbol{z}[:, t] = \boldsymbol{H}[:, t] * \mathbf{u}[:, t]$
3: In parallel for $0 \le t < D$ : $\boldsymbol{y}[:, t] \leftarrow (\boldsymbol{x}[:, t] + \boldsymbol{B}_1[:, t]) \odot (\boldsymbol{z}[:, t] + \boldsymbol{B}_2[:, t])$. ▷ See eq. (18)
4: **return** $\boldsymbol{y}$

**Remark D.21.** The definition of a BASECONV layer in Equation (19) has the input go through a linear layer before the convolution operation. For this section we will assume the linear layer is the identity matrix, as it is not needed for the results in this section.

**Assumption D.22.** Moving forward we assume the weight matrix $\boldsymbol{W} \in \mathbb{R}^{D \times D}$ in Definition D.20 also has the property $\boldsymbol{W} \in (\mathcal{B}\mathcal{B}^*)^{\text{poly-}\log D}_{\text{poly-}\log D}$. Consequently, each matrix $\boldsymbol{W}$ has $\tilde{\mathcal{O}}(D)$ parameters and runtime for matrix vector multiplication Dao et al. (2020).

In this section, we will establish some additional basic primitives that we expect need to implement via a BASECONV layer: `shift` and `remember`. We specify them below:

**Definition D.23.** `shift`$(\boldsymbol{y}, r, t, f)$
Shift an sequential input of length $N$ up or down by $s$ entries:
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N \times D}$, $s \ge 0$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N \times D}$ where $\boldsymbol{z}^+ =$ `shift_down`$(\boldsymbol{y}, s)$ and $\boldsymbol{z}^- =$ `shift_up`$(\boldsymbol{y}, s)$

$$\boldsymbol{y} \equiv \begin{pmatrix} \leftarrow \boldsymbol{y}_0 \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{y}_{i-1} \rightarrow \\ \leftarrow \boldsymbol{y}_i \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{y}_{N-1} \rightarrow \end{pmatrix} \quad \boldsymbol{z}^+ \equiv \begin{pmatrix} \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \\ \leftarrow \boldsymbol{y}_0 \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{y}_{N-1-s} \rightarrow \end{pmatrix} \quad \boldsymbol{z}^- \equiv \begin{pmatrix} \leftarrow \boldsymbol{y}_s \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{y}_{N-1} \rightarrow \\ \leftarrow \mathbf{0} \rightarrow \\ \vdots \\ \leftarrow \mathbf{0} \rightarrow \end{pmatrix}$$

The following proposition is defining the convolution Kernel that computes the $\texttt{shift\_down}\left(\cdot, \lfloor \frac{N}{2} \rfloor\right)$ primitive:

**Proposition D.24.** *Define* $\boldsymbol{H} \in \mathbb{R}^{2N \times D}$ *as*

$$\boldsymbol{H}[k,:] = \begin{cases} \mathbf{1}^D & \text{if } k = N \\ 0 & \text{otherwise} \end{cases}.$$

*For any* $\mathbf{u} \in \mathbb{R}^{2N \times D}$, $\boldsymbol{H} * \mathbf{u}$ *will result in*

$$\boldsymbol{H} * \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{u}_1 \end{pmatrix},$$

*where* $\mathbf{u}_1, \mathbf{u}_2 \in \mathbb{R}^{N \times D}$.

*Proof.* The convolution operation: $\boldsymbol{H} * \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}$ where each column of $\boldsymbol{H}$ is convolved with each column of $\mathbf{u}$ can be restated as a polynomial multiplication. For column i, $0 \leq i < 2N$,

$$\boldsymbol{H}[:,i] * \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}[:,i] = \text{coeff}((X^N \cdot \mathbf{u}[:,i](X)) \mod X^{2N}).$$

Note that the columns of $\boldsymbol{H}$ are all $\mathbf{e}_N$ basis vectors and $\text{poly}(\mathbf{e}_N) = X^N$.

When we multiply the term through the input polynomial we get,

$$\begin{aligned} &\text{coeff}\left(X^N \cdot \left(\mathbf{u}[0][i] + \mathbf{u}[1][i]X + \cdots + \mathbf{u}[2N-1][i]X^{2N-1}\right) \mod X^{2N}\right) \\ &= \text{coeff}(\mathbf{u}[0][i]X^N + \mathbf{u}[1][i]X^{N+1} + \cdots + \mathbf{u}[2N-1][i]X^{3N-1} \mod X^{2N}). \end{aligned}$$

With the lower order terms all becoming zeros, the above is same as

$$\begin{aligned} &\text{coeff}((0 + 0X + \cdots 0X^{N-1} \\ &+ \mathbf{u}[0][i]X^N + \mathbf{u}[1][i]X^{N+1} + \cdots + \mathbf{u}[2N-1][i]X^{3N-1}) \mod X^{2N}). \end{aligned}$$

After we take the $\mod X^{2N}$ we get

$$\text{coeff}(0 + 0X + \cdots + 0X^{N-1} + \mathbf{u}[0][i]X^N + \cdots + \mathbf{u}[N-1][i]X^{2N-1}),$$

which implies that $\boldsymbol{H} * \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}$ is

$$\begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{u}_1 \end{pmatrix},$$

as desired. □

We also define the following primitive:

**Definition D.25.** `remember`$(\boldsymbol{y}, r, t, f)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}, r \in \mathbb{Z}, t \in \mathbb{Z}, f : \mathbb{R}^{t-r} \to \mathbb{R}^{t-r+s}, \boldsymbol{v}_1 \in \mathbb{R}^r, \boldsymbol{x} \in \mathbb{R}^{t-r}$, where $\boldsymbol{y}$ is defined as below.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'}$, which is defined as follows:

$$\boldsymbol{y} \equiv \begin{pmatrix} \leftarrow \boldsymbol{v}_1 \rightarrow \\ \hline \leftarrow \boldsymbol{x} \rightarrow \\ \hline \mathbf{0}^{s \times d'} \\ \hline \leftarrow \boldsymbol{v}_2 \rightarrow \\ \hline \mathbf{0} \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix} \qquad \boldsymbol{z} \equiv \begin{pmatrix} \leftarrow \boldsymbol{v_1} \rightarrow \\ \hline \leftarrow f(\boldsymbol{x}) \rightarrow \\ \hline \leftarrow \boldsymbol{v}_2 \rightarrow \\ \hline \mathbf{0} \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix}$$

We will need the following BASECONV implementation of `remember`:

**Proposition D.26** (Arora et al. (2024), The Remembering Primitive)**.** *For any $\boldsymbol{x} \in \mathbb{R}^{n \times d'}, \boldsymbol{v}_1 \in \mathbb{R}^{r \times d'}, \boldsymbol{v}_2 \in \mathbb{R}^{m-r}$ where $n = t - r$ contained in some $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$ such that $\boldsymbol{v}_1$ is in the first $r$ rows, $\boldsymbol{x}$ is in the next $n$ rows, 0s fill up the next $s$ rows, and $\boldsymbol{v}_2$ are in the next $m - r$ rows, for some $3n + 3m + 2s + 2t \leq N'$ so that for $\boldsymbol{h} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$ with $\boldsymbol{x} * \boldsymbol{h} \in \mathbb{R}^{(n+s) \times d'}$ and $\boldsymbol{v} * \boldsymbol{h} \in \mathbb{R}^{(m+t) \times d'}$, where $\boldsymbol{v} \in \mathbb{R}^{m \times d'}$ is defined as $\boldsymbol{v}_2 + $ `shift_down` $(\boldsymbol{v}_1, m - r)$, there exists a $(N', 8, d', N', d') -$ BASECONV that computes `remember`$(\boldsymbol{y}, r, t, f)$, where $f$ can be implemented in 1 layer of* BASECONV *through the parameters $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}, \boldsymbol{h} \in \mathbb{R}^{N' \times d'}, \boldsymbol{b}_1 \in \mathbb{R}^{N' \times d'}, \boldsymbol{b}_2 \in \mathbb{R}^{N' \times d'}$ as defined below:*

$$f(\boldsymbol{u}) = \left( \begin{pmatrix} \boldsymbol{u}\boldsymbol{W} \\ \mathbf{0}^{s \times d'} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1 \\ \mathbf{1}^{s \times d'} \end{pmatrix} \right) \odot \left( \boldsymbol{u} * \boldsymbol{h} + \begin{pmatrix} \boldsymbol{b}_2 \\ \mathbf{0}^{s \times d'} \end{pmatrix} \right)$$

We will also need the following generalization of the above result:

**Corollary D.27** (Arora et al. (2023))**.** *Let $\boldsymbol{y}$ be as in Proposition D.26 but now let $f$ be implemented with* BASECONV$(N, L, D, N, D)$. *Then* `remember`$(\boldsymbol{y}, r, t, f)$ *where $t - r = n$ can be implemented with* BASECONV *via $(N, O(L), D, N, D) -$* BASECONV.

The rest of Appendix D will use this $5-$tuple notation for BASECONV:

**Definition D.28.** Lets define a 5-tuple notation for a BASECONV layer as $(N, \ell, D, N', D') -$ BASECONV with $\ell$ layers such that:

1. Input and output are $N \times D$ matrices.
2. Each layer is defined by Definition D.20 where $N$ and $D$ are replaced by $N'$ and $D'$. I.e. each layer takes in $N' \times D'$ matrices and output $N' \times D'$ matrices. We refer to the tuple $(N', D')$ as the *inner dimension* of the model.
3. The matrices are projected from $(N, D) \to (N', D')$ (and vice-versa) via a linear projection.

We state the following bounds on parameters and runtime for a single BASECONV layer:

**Proposition D.29** (Arora et al. (2023))**.** *An $(N, 1, D, N, D) -$ BASECONV requires $\tilde{O}(ND)$ parameters and runtime.*

We state the following result that says arithmetic circuit can be represented as a BASECONV model:

**Theorem D.30** (Arora et al. (2023), Theorem H.21)**.** *For any $(ND, s, \Delta, w)$-arithmetic circuit $\mathcal{C}$, there exists an equivalent $(N, \Delta', D, N', D') -$ BASECONV with $\Delta' = \mathcal{O}(\Delta \log w)$, $N' = \mathcal{O}(w)$, $D' = D$ that simulates $\mathcal{C}$.*

### D.2 PRIMITIVES

In this section, we provide theoretical results about primitives.

- In Appendix D.2.1, we implement the three primitives (READ, LINEAR, and MULTIPLY) from Section 3.3 using BASECONV, each using a single layer.
- Next, in Appendix D.2.2 and D.2.3, we briefly sketch how the three primitives READ, LINEAR, and MULTIPLY can be used in composition to exactly express gradient descent and Newton's method iterations on least squares (see Appendix A).
- Finally, in Appendix D.2.4, we provide a proof that a single layer of causal softmax attention cannot exactly represent the entry-wise squaring function. As a corollary, since entry-wise square is a special case of MULTIPLY, this implies that attention cannot exactly express the MULTIPLY task for all arguments.

**BASECONV parameterization** We recount the parameterization of BASECONV from Equation 2:

$$\begin{aligned} \boldsymbol{y} &:= \left( \underbrace{(\boldsymbol{u} \cdot \boldsymbol{W}_{gate} + \boldsymbol{b}_{gate})}_{\textbf{Linear Projection}} \odot \underbrace{(\boldsymbol{h} * (\boldsymbol{u} \cdot \boldsymbol{W}_{in} + \boldsymbol{b}_{in}) + \boldsymbol{b}_{conv})}_{\textbf{Convolution}} \right) \cdot \boldsymbol{W}_{out} + \boldsymbol{b}_{out} \\ &:= W_{out}(W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u}))) \end{aligned} \tag{19}$$

where $W_{in}, W_{gate}, W_{out}$ are linear projections $\mathbb{R}^D \to \mathbb{R}^D$.

#### D.2.1 1-LAYER BASECONV CAN IMPLEMENT LINEAR ALGEBRA PRIMITIVES

Below, we formally define the linear algebra primitives we discuss in Section 3.3, and we describe our BASECONV weight constructions.

**Read** The READ operator, which maps inputs $\mathbf{u} \in \mathbb{R}^{N \times d}$ to outputs $\mathbf{y} \in \mathbb{R}^{N \times d}$, is:

$$\text{READ}(i, j, a, b)(\mathbf{u}) = \begin{cases} \mathbf{u}[k, a:b] & k \neq j \\ \mathbf{u}[i, a:b] & k = j \end{cases}. \tag{20}$$

Our implementation requires the use of the positional encodings and residual connections within the BASECONV architecture. Concretely, consider the input

$$\boldsymbol{u}_{in} = \left( \begin{array}{cccc} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \dots & \boldsymbol{e}_N \\ \hline \boldsymbol{u}[1,:] & \boldsymbol{u}[2,:] & \dots & \boldsymbol{u}[N,:] \end{array} \right),$$

where the basis vector $\boldsymbol{e}_k$ represents the positional encoding for the $k$-th entry of the sequence. Define the output of the BASECONV layer *with residual connection*:

$$\boldsymbol{y} := W_{out}(W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u})) + \boldsymbol{u}).$$

Then the following weight construction is equivalent to $\text{READ}(i, j, a, b)$:

- $W_{gate}(\boldsymbol{u}[k,:]) := \boldsymbol{u}[k, j]\mathbf{1}^D$
- $Conv(W_{in}(\boldsymbol{u}))[k,:] := \boldsymbol{u}[k + i - j, :] - \boldsymbol{u}[k,:]$
- $W_{out} := proj(a:b)$.

In particular, $W_{gate}$ is defined such that

$$W_{gate}(\boldsymbol{u}[k,:]) = \begin{cases} \mathbf{1}^D & k = j \\ \mathbf{0}^D & k \neq j \end{cases}.$$

Thus

$$W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u})) = \begin{cases} \boldsymbol{u}[k + i - j, :] - \boldsymbol{u}[k,:] = \boldsymbol{u}[i,:] - \boldsymbol{u}[j,:] & k = j \\ \mathbf{0}^D & k \neq j \end{cases}.$$

Finally,

$$W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u})) + \boldsymbol{u} = \begin{cases} \boldsymbol{u}[i,:] & k = j \\ \boldsymbol{u}[k,:] & k \neq j \end{cases}$$

so the final output of this layer will be exactly equivalent to $\text{READ}(i, j, a, b)$.

**Linear transformation** The LINEAR operator, which maps inputs $\mathbf{u} \in \mathbb{R}^{N \times d_{in}}$ to outputs $\mathbf{y} \in \mathbb{R}^{N \times d_{out}}$, is:

$$\text{LINEAR}(\boldsymbol{H})(\mathbf{u}) = \mathbf{u}\boldsymbol{H} \tag{21}$$

where $\boldsymbol{H} : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ is a linear map.

Define $Conv(W_{in}(\mathbf{u})) = \mathbf{1}_D$, $W_{gate} = I$, and $W_{out} = \boldsymbol{H}$. Then

$$W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u})) = \boldsymbol{u}$$

so

$$W_{out}(W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u}))) = \boldsymbol{u}\boldsymbol{H}.$$

Thus the output of this layer is exactly equivalent to LINEAR($\boldsymbol{H}$).

**Element-wise multiply** The MULTIPLY operator, which maps inputs $\mathbf{u} \in \mathbb{R}^{N \times d_{in}}$ to outputs $\mathbf{y} \in \mathbb{R}^{N \times d_{out}}$, is:

$$\text{MULTIPLY}(a, b, d_{out})(\mathbf{u}) = \mathbf{u}[:, a : a + d_{out}] \odot \mathbf{u}[:, b : b + d_{out}] \tag{22}$$

Define $Conv =$ Identity, $W_{in} = proj(a : a + d_{out})$, $W_{gate} = proj(b : b + d_{out})$, and $W_{out} = \boldsymbol{I}$.

Then

$$W_{gate}(\mathbf{u}) \odot Conv(W_{in}(\mathbf{u})) = \mathbf{u}[:, a : a + d_{out}] \odot \mathbf{u}[:, b : b + d_{out}].$$

Since $W_{out} = \boldsymbol{I}$, the output of this layer will be equivalent to MULTIPLY$(a, b, d_{out})$.

### D.2.2 GRADIENT DESCENT

We assume our input is of the form

$$\boldsymbol{u} = \begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{x}_0 \\ b_1 & \dots & b_N & 0 \end{pmatrix}.$$

Our goal is to compute the gradient update

$$\boldsymbol{x}_1 := \boldsymbol{x}_0 - \eta \sum_{i=1}^{N} (\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i)\boldsymbol{a}_i. \tag{23}$$

Intuitively, our argument proceeds similarly to the causal gradient descent construction from Appendix D.3.1:

- First, we repeatedly apply READ and LINEAR to move the information $\{\boldsymbol{a}_i, b_i\}\, \forall i$ into e.g. the final entry of the sequence. Without loss of generality, we omit the rest of the sequence, and assume we have access to a large enough embedding dimension that we can make use of arbitrary amounts of memory.
  After this phase, our $\boldsymbol{u}$ is of the form
  $$\dots (\boldsymbol{x}_0 \quad 0 \quad \boldsymbol{a}_1 \quad \dots \quad \boldsymbol{a}_N \quad b_1 \quad \dots \quad b_N \quad \dots)^T.$$
- Next, we use MULTIPLY and LINEAR to compute and store $\{\boldsymbol{x}_0^T \boldsymbol{a}_i\}$ for all $i$. We will end up with
  $$\boldsymbol{u} = \dots \begin{pmatrix} \boldsymbol{x}_0 & 0 & \{\boldsymbol{a}_i\}_i & \{b_i\}_i & \{\boldsymbol{x}_0^T \boldsymbol{a}_i\}_i & \dots \end{pmatrix}.$$
- We use LINEAR to compute and store $\{\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i\}$ for all $i$:
  $$\boldsymbol{u} = \dots \begin{pmatrix} \boldsymbol{x}_0 & 0 & \{\boldsymbol{a}_i\}_i & \{b_i\}_i & \{\boldsymbol{x}_0^T \boldsymbol{a}_i\}_i & \{\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i\}_i & \dots \end{pmatrix}.$$
- We use MULTIPLY and LINEAR to compute and store $\{(\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i)\boldsymbol{a}_i\}$ for all $i$:
  $$\boldsymbol{u} = \dots \begin{pmatrix} \boldsymbol{x}_0 & 0 & \{\boldsymbol{a}_i\}_i & \{b_i\}_i & \{\boldsymbol{x}_0^T \boldsymbol{a}_i\}_i & \{(\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i)\boldsymbol{a}_i\}_i & \dots \end{pmatrix}.$$
- Finally, we can use LINEAR to compute the gradient update:
  $$\boldsymbol{u} = \dots \begin{pmatrix} \boldsymbol{x}_0 - \eta \sum_{i=1}^{N} (\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i)\boldsymbol{a}_i & 0 & \{\boldsymbol{a}_i\}_i & \{b_i\}_i & \{\boldsymbol{x}_0^T \boldsymbol{a}_i\}_i & \{(\boldsymbol{x}_0^T \boldsymbol{a}_i - b_i)\boldsymbol{a}_i\}_i & \dots \end{pmatrix}.$$

### D.2.3 NEWTON'S METHOD

We assume our input is of the form

$$\boldsymbol{u} = \begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{M}_0[1,:] & \dots & \boldsymbol{M}_0[D,:] \\ b_1 & \dots & b_N & 0 & \dots & 0 \end{pmatrix}.$$

Our goal is to compute the Newton's iterate:

$$\boldsymbol{M}_1 := \boldsymbol{M}_0(2\boldsymbol{I} - (\boldsymbol{a}^T\boldsymbol{a})\boldsymbol{M}_0), \tag{24}$$

where

$$\boldsymbol{a} = \begin{pmatrix} \leftarrow \boldsymbol{a}_1 \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{a}_N \rightarrow \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}. \tag{25}$$

For any matrix $\boldsymbol{M} \in \mathbb{R}^{n\times p}$, let $flt$ denote the `flatten` operation, so that $flt(\boldsymbol{M})$ represent a vectorized version of $\boldsymbol{M}$: $flt(\boldsymbol{M}) \in \mathbb{R}^{np}$.

We proceed similarly to the argument from Appendix D.2.2.

- First, we repeatedly apply READ and LINEAR to move all information $\{\boldsymbol{a}_i\}_i \, \forall i$ and $flt(\boldsymbol{M})$ to e.g. the final entry of the sequence. We omit the rest of the sequence for notational ease, and we assume we have access to a large enough embedding dimension that we can make use of arbitrary amounts of memory.
  After this phase, we have
  $$\boldsymbol{u} = \dots (flt(\boldsymbol{M}_0) \quad \{\boldsymbol{a}_i\}_i \quad \dots).$$
- Using LINEAR, we can copy and rearrange the $\boldsymbol{a}_i$'s to construct copies of $flt(\boldsymbol{a})$ and $flt(\boldsymbol{a}^T)$:
  $$\boldsymbol{u} = \dots \big(flt(\boldsymbol{M}_0) \quad \{\boldsymbol{a}_i\}_i \quad flt(\boldsymbol{a}^T) \quad flt(\boldsymbol{a}) \quad \dots\big).$$
- Now, note that we can represent the matrix multiplication $\boldsymbol{a}^T\boldsymbol{a}$ as a linear combination of the entries of the element-wise multiplication $flt(\boldsymbol{a}^T) \odot flt(\boldsymbol{a})$. This means that we can obtain $flt(\boldsymbol{a}^T\boldsymbol{a})$ using a single application of MULTIPLY and LINEAR:
  $$\boldsymbol{u} = \dots \big(flt(\boldsymbol{M}_0) \quad \{\boldsymbol{a}_i\}_i \quad flt(\boldsymbol{a}^T) \quad flt(\boldsymbol{a}) \quad flt(\boldsymbol{a}^T\boldsymbol{a}) \dots\big).$$
- By the same argument, we can obtain $flt((\boldsymbol{a}^T\boldsymbol{a})\boldsymbol{M}_0)$ using another application of MULTIPLY and LINEAR:
  $$\boldsymbol{u} = \dots \big(flt(\boldsymbol{M}_0) \quad \{\boldsymbol{a}_i\}_i \quad flt(\boldsymbol{a}^T) \quad flt(\boldsymbol{a}) \quad flt((\boldsymbol{a}^T\boldsymbol{a})\boldsymbol{M}_0) \dots\big).$$
- Finally, we have that $flt(\boldsymbol{M}_1) := 2flt(\boldsymbol{M}_0) - flt((\boldsymbol{a}^T\boldsymbol{a})\boldsymbol{M}_0)$ can be obtained using LINEAR once more:
  $$\boldsymbol{u} = \dots \big(flt(\boldsymbol{M}_1) \quad \{\boldsymbol{a}_i\}_i \quad flt(\boldsymbol{a}^T) \quad flt(\boldsymbol{a}) \quad flt((\boldsymbol{a}^T\boldsymbol{a})\boldsymbol{M}_0) \dots\big).$$

### D.2.4 SOFTMAX ATTENTION CAN'T IMPLEMENT ELEMENT-WISE SQUARING.

In this section, we consider the following parameterization of *softmax attention*:

$$\text{Attn}(\boldsymbol{u}) = \text{softmax}\left((\boldsymbol{u}\boldsymbol{W_Q})(\boldsymbol{u}\boldsymbol{W_K})^T + \boldsymbol{M}\right)(\boldsymbol{u}\boldsymbol{W_V} + \boldsymbol{B}), \tag{26}$$

where $\boldsymbol{u} \in \mathbb{R}^{N\times D}$, $\boldsymbol{W_Q}, \boldsymbol{W_K}, \boldsymbol{W_V} \in \mathbb{R}^{D\times D}$, $\boldsymbol{B} \in \mathbb{R}^{N\times D}$, and $\boldsymbol{M} \in \mathbb{R}^{N\times N}$ is the causal attention mask:

$$\boldsymbol{M}_{ij} = \begin{cases} -\infty & i < j \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

**Theorem D.31.** *One-layer single-headed causal softmax attention cannot exactly represent the entry-wise squaring function* $\text{SQUARE} : \mathbb{R}^{N\times D} \to \mathbb{R}^{N\times D}$ *s.t.*

$$\text{SQUARE}(\boldsymbol{u})_{ij} = \boldsymbol{u}_{ij}^2$$

*for all* $\boldsymbol{u} \in \mathbb{R}^{N\times D}$.

*Proof.* We proceed by contradiction. Let's assume there exists $\boldsymbol{W_Q}, \boldsymbol{W_K}, \boldsymbol{W_V}, \boldsymbol{B} \in \mathbb{R}^{D\times D}$ and $\boldsymbol{B} \in \mathbb{R}^{N\times D}$ such that $\forall \boldsymbol{u} \in \mathbb{R}^{N\times D}$,

$$\text{softmax}\left((\boldsymbol{u}\boldsymbol{W_Q})(\boldsymbol{u}\boldsymbol{W_K})^T + \boldsymbol{M}\right)(\boldsymbol{u}\boldsymbol{W}_V + \boldsymbol{B}) = \text{SQUARE}(\boldsymbol{u}). \tag{28}$$

Consider the set of inputs $\boldsymbol{u} \in \mathbb{R}^{N\times D}$ with at most one non-zero entry, defined as

$$\boldsymbol{u}_{ij} = \begin{cases} \boldsymbol{u}_{ij} & (i,j) = (a,b) \\ 0 & \text{else} \end{cases} \tag{29}$$

for an arbitrary choice of $a \in [N]$, $b \in [D]$. Then:

$$\boldsymbol{Q} := \boldsymbol{u}\boldsymbol{W_Q} = \begin{pmatrix} \mathbf{0}^N \\ \hline \vdots \\ \hline \mathbf{0}^N \\ \hline \boldsymbol{u}_{ab}\boldsymbol{W_Q}[b,:] \\ \hline \vdots \\ \hline \mathbf{0}^N \end{pmatrix} \tag{30}$$

where $\boldsymbol{Q}$'s rows are all $\mathbf{0}$ except for the $a$-th, which is $\boldsymbol{u}_{ab}\boldsymbol{W_Q}[b,:]$.

Similarly:

$$\boldsymbol{K} := \boldsymbol{u}\boldsymbol{W_K} = \begin{pmatrix} \mathbf{0}^N \\ \hline \vdots \\ \hline \mathbf{0}^N \\ \hline \boldsymbol{u}_{ab}\boldsymbol{W_K}[b,:] \\ \hline \vdots \\ \hline \mathbf{0}^N \end{pmatrix} \tag{31}$$

and

$$\boldsymbol{V} := \boldsymbol{u}\boldsymbol{W}_{\boldsymbol{V}} = \begin{pmatrix} \mathbf{0}^N \\ \hline \vdots \\ \hline \mathbf{0}^N \\ \hline \boldsymbol{u}_{ab}\boldsymbol{W}_{\boldsymbol{V}}[b,:] \\ \hline \vdots \\ \hline \mathbf{0}^N \end{pmatrix} \tag{32}$$

Then the pre-softmax attention matrix, $\boldsymbol{A}' = \boldsymbol{Q}\boldsymbol{K}^T$, satisfies

$$\boldsymbol{A}'_{ij} = \begin{cases} \boldsymbol{u}_{ab}^2(\boldsymbol{W}_{\boldsymbol{Q}}\boldsymbol{W}_{\boldsymbol{K}}^T)_{bb} & (i,j) = (a,a) \\ 0 & \text{otherwise} \end{cases}. \tag{33}$$

Define

$$C := \boldsymbol{u}_{ab}^2(\boldsymbol{W}_{\boldsymbol{Q}}\boldsymbol{W}_{\boldsymbol{K}}^T)_{bb}. \tag{34}$$

Now consider what happens after we apply the softmax operator. Recall that the softmax operator is defined as

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(\boldsymbol{z}_i)}{\sum_{j=1}^D \exp(\boldsymbol{z}_j)} \tag{35}$$

for $\boldsymbol{z} \in \mathbb{R}^D$. Then $\boldsymbol{A} := \text{softmax}(\boldsymbol{A}' + \boldsymbol{M})$ satisfies

$$\boldsymbol{A}_{ij} = \begin{cases} \frac{1}{i} & i \neq a \\ \frac{1}{\exp(C)+a-1} & i = a,\ j \neq a \\ \frac{\exp(C)}{\exp(C)+a-1} & (i,j) = (a,a) \end{cases} \tag{36}$$

Now let's consider the output of softmax attention:

$$\boldsymbol{O} = \boldsymbol{A}(\boldsymbol{V} + \boldsymbol{B}) \tag{37}$$

such that $\boldsymbol{O} = \text{SQUARE}(\boldsymbol{u})$.

Note that for $i \neq a$:

$$\boldsymbol{O}[i,:] = \frac{1}{i}\sum_{k=1}^{i}(\boldsymbol{V} + \boldsymbol{B})[k,:] \tag{38}$$

and this must also be equal to $\mathbf{0}^N = \text{SQUARE}(\boldsymbol{u})[i,:]$. We consider three cases:

- First, consider $i < a$ in order from $i = 1, \ldots, a-1$. Since this equality is true for all $i < a$, we can verify that $(\boldsymbol{V} + \boldsymbol{B})[i,:]$ must equal $\mathbf{0}^N$ for all $i < a$.
- Next, looking at $i = a + 1$, we have

  $$\frac{1}{a+1}\left((\boldsymbol{V} + \boldsymbol{B})[a,:] + (\boldsymbol{V} + \boldsymbol{B})[a+1,:]\right) = \mathbf{0}^N \tag{39}$$

  so we must have

  $$(\boldsymbol{V} + \boldsymbol{B})[a,:] = -(\boldsymbol{V} + \boldsymbol{B})[a+1,:] \tag{40}$$

- Finally, from $i \geq a + 1$, we can again conclude that $(\boldsymbol{V} + \boldsymbol{B})[i,:]$ must equal $\mathbf{0}^N$ for all $i > a + 1$.

This means the only rows of $\boldsymbol{V}+\boldsymbol{B}$ that might not be zero are $(\boldsymbol{V}+\boldsymbol{B})[a,:]$ and $(\boldsymbol{V}+\boldsymbol{B})[a+1,:]$. Thus looking at the $a$-th row:

$$\frac{\exp(C)}{\exp(C)+a-1}(\boldsymbol{V}+\boldsymbol{B})[a,:] = \textsc{Square}(\boldsymbol{u})[a,:]$$
$$= \begin{bmatrix} 0 & \dots & \boldsymbol{u}_{ab}^2 & \dots & 0 \end{bmatrix}$$

Recall that from above,

$$\boldsymbol{V}[a,:] = \boldsymbol{u}_{ab}\boldsymbol{W}_{\boldsymbol{V}}[b,:] \tag{41}$$

Then analyzing entry-wise, we have:

$$\frac{\exp(C)}{\exp(C)+a-1}\left(\boldsymbol{u}_{ab}\boldsymbol{W}_{\boldsymbol{V}}[b,j]+\boldsymbol{B}[a,j]\right)=0 \tag{42}$$

for all $j \neq b$, and

$$\frac{\exp(C)}{\exp(C)+a-1}\left(\boldsymbol{u}_{ab}\boldsymbol{W}_{\boldsymbol{V}}[b,b]+\boldsymbol{B}[a,b]\right)=\boldsymbol{u}_{ab}^2. \tag{43}$$

We now plug back in our expression for $C$ and simplifying the latter equation. For ease of notation, denote $A := (\boldsymbol{W}_{\boldsymbol{Q}}\boldsymbol{W}_{\boldsymbol{K}}^T)_{bb}$, $V := \boldsymbol{W}_{\boldsymbol{V}}[b,b]$, and $B := \boldsymbol{B}[a,b]$. Then the expression simplifies to:

$$V\exp(A\boldsymbol{u}_{ab}^2)\boldsymbol{u}_{ab}+B\exp(A\boldsymbol{u}_{ab}^2)=\exp(A\boldsymbol{u}_{ab}^2)\boldsymbol{u}_{ab}^2+(a-1)\boldsymbol{u}_{ab}^2$$

This must hold for all non-zero values of $\boldsymbol{u}_{ab}$. We can take $V=B=0$, but we are still left with

$$-\exp(A\boldsymbol{u}_{ab}^2)\boldsymbol{u}_{ab}^2=(a-1)\boldsymbol{u}_{ab}^2$$
$$-\exp(A\boldsymbol{u}_{ab}^2)=a-1$$

However, there is no choice of $A$ such that this statement holds. This completes the proof by contradiction.

As a corollary, we have

**Corollary D.32.** *One-layer single-headed causal softmax attention cannot exactly represent the entry-wise multiply function* $\textsc{Multiply}:\mathbb{R}^{N\times D}\to\mathbb{R}^{N\times d_{out}}$ *s.t.*

$$\textsc{Multiply}(a,b,d_{out})(\mathbf{u}) = \mathbf{u}[:,a:a+d_{out}]\odot\mathbf{u}[:,b:b+d_{out}] \tag{44}$$

*for all* $\boldsymbol{u}\in\mathbb{R}^{N\times D}$ *and all choices of* $a$, $b$, $d_{out}$.

*Proof.* Note that for $a=0$, $b=0$, and $d_{out}=D$,

$$\textsc{Square}(\boldsymbol{u}) = \textsc{Multiply}(a,b,d_{out})(\boldsymbol{u}).$$

Since softmax attention cannot exactly represent $\textsc{Square}$ for all $\boldsymbol{u}$, it also cannot represent $\textsc{Multiply}$ for all $\boldsymbol{u}$. □

### D.3 Upper and lower bounds with BaseConv for gradient descent

In this section, we detail upper and lower bounds for implementing gradient descent using BaseConv, as discussed in Section 4.1.

- **Upper bounds.** We provide two explicit constructions for implementing iterations gradient descent on linear regression: one for *non-causal* BaseConv requiring $O(1)$ layers and $O(D)$ state size, and one for *causal* BaseConv requiring $O(1)$ layers and $O(D^2)$ state size.
- **Lower bounds.** In Appendix D.3.2, we prove that our constructions are asymptotically optimal with respect to layers and state size.

#### D.3.1 Upper bounds: BaseConv can implement gradient descent for linear regression

In this section, we provide weight constructions for exactly implementing gradient descent on linear regression. Recall:

$$\mathcal{L}_N = \frac{1}{2N}\sum_{i=1}^{N}(\boldsymbol{x}^T\boldsymbol{a}_i - \boldsymbol{b}_i)^2 \tag{45}$$

so

$$\nabla_{\boldsymbol{x}}\mathcal{L}_N = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}^T\boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \tag{46}$$

$$= \frac{1}{N}\left(\sum_{i=1}^{N}\boldsymbol{b}_i\boldsymbol{a}_i - \left(\sum_{i=1}^{N}\boldsymbol{a}_i\boldsymbol{a}_i^T\right)\boldsymbol{x}\right) \tag{47}$$

**Non-causal BaseConv** This weight construction uses Equation 46 to compute the gradient descent update.

We note that non-causal constructions for in-context linear regression are standard in the literature: e.g. Von Oswald et al. (2023); Ahn et al. (2024).

We start with input:

$$\boldsymbol{b} \equiv \left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{a}_q \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \end{array}\right)$$

We define the initial embedding:

$$\left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \boldsymbol{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \boldsymbol{x}_0 & \dots & \boldsymbol{x}_0 & \boldsymbol{x}_0 \\ \hline \boldsymbol{0}^D & \dots & \boldsymbol{0}^D & \boldsymbol{0}^D \\ \hline \boldsymbol{0}^D & \dots & \boldsymbol{0}^D & \boldsymbol{0}^D \\ \hline\hline \boldsymbol{0}^D & \dots & \boldsymbol{0}^D & \boldsymbol{a}_q \\ \hline 0 & \dots & 0 & 0 \end{array}\right)$$

We drop the bottom two rows of the block matrix representation for now and show how to perform the gradient descent update with the rest of the embedding.

Layer 1:

$$\underbrace{\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \end{pmatrix}}_{conv(in\_proj(\cdot))} \odot \underbrace{\begin{pmatrix} \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \mathbf{1} \rightarrow \\ \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \end{pmatrix}}_{gate\_proj(\cdot)} = \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \end{pmatrix}$$

$$\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \end{pmatrix} \underbrace{\rightarrow}_{out\_proj(\cdot)} \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\mathbf{1}^D \rightarrow \end{pmatrix}$$

Layer 2:

$$\underbrace{\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\mathbf{1}^D \rightarrow \end{pmatrix}}_{conv(in\_proj(\cdot))} \odot \underbrace{\begin{pmatrix} \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \mathbf{1} \rightarrow \\ \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \boldsymbol{a}_i \rightarrow \end{pmatrix}}_{gate\_proj(\cdot)} = \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \rightarrow \end{pmatrix}$$

$$\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \rightarrow \end{pmatrix} \underbrace{\rightarrow}_{out\_proj(\cdot)=Identity} \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \rightarrow \end{pmatrix}$$

Layer 3:

$$\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \rightarrow \end{pmatrix} \underbrace{\rightarrow}_{conv(in\_proj(\cdot))} \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow \sum_{i=1}^{N} (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i \rightarrow \end{pmatrix}$$

$$\begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 \rightarrow \\ \leftarrow \boldsymbol{a}_i \odot \boldsymbol{x}_0 \rightarrow \\ \leftarrow \underbrace{\sum_{i=1}^{N} (\boldsymbol{x}_0^T \boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{a}_i}_{=\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}_0)} \rightarrow \end{pmatrix} \underbrace{\rightarrow}_{out\_proj(\cdot)} \begin{pmatrix} \leftarrow \boldsymbol{a}_i \rightarrow \\ \leftarrow \boldsymbol{b}_i \rightarrow \\ \leftarrow \boldsymbol{x}_0 - \eta \nabla_{\boldsymbol{x}} \mathcal{L}(\mathbf{w_0}) \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \\ \leftarrow \mathbf{0}^D \rightarrow \end{pmatrix}$$

**Causal BASECONV** This weight construction uses Equation 47 to compute the gradient descent update.

We start with input:

$$\boldsymbol{b} \equiv \begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \end{pmatrix}$$

We use two BASECONV layers to construct an initial embedding, after which each gradient descent update step will only require a single BASECONV layer.

In the following construction, we use $flt$ to denote the `flatten` operation, which maps an $M \times N$ matrix to a $MN$-entry vector with the same elements.

Layer 1:

$$\underbrace{\begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ flt(\boldsymbol{a}_1(\mathbf{1}^D)^T) & \dots & flt(\boldsymbol{a}_N(\mathbf{1}^D)^T) & flt(\mathbf{0}^D(\mathbf{0}^D)^T) \end{pmatrix}}_{conv(in\_proj(\cdot))} \odot \underbrace{\begin{pmatrix} \leftarrow \mathbf{1}^D \rightarrow \\ \leftarrow \mathbf{1} \rightarrow \\ \leftarrow \mathbf{1}^D \rightarrow \\ \boldsymbol{b}_1\mathbf{1}^D & \dots & \boldsymbol{b}_N\mathbf{1}^D & \mathbf{0}^D \\ flt(\mathbf{1}^D\boldsymbol{a}_1^T) & \dots & flt(\mathbf{1}^D\boldsymbol{a}_N^T) & flt(\mathbf{0}^D(\mathbf{0}^D)^T) \end{pmatrix}}_{gate\_proj(\cdot)} =$$

$$\begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \boldsymbol{b}_1\boldsymbol{a}_1 & \dots & \boldsymbol{b}_1\boldsymbol{a}_N & \mathbf{0}^D \\ flt(\boldsymbol{a}_1\boldsymbol{a}_1^T) & \dots & flt(\boldsymbol{a}_N\boldsymbol{a}_N^T) & flt(\mathbf{0}^D(\mathbf{0}^D)^T) \end{pmatrix} \underbrace{\rightarrow}_{out\_proj=Identity} \begin{pmatrix} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \boldsymbol{b}_1\boldsymbol{a}_1 & \dots & \boldsymbol{b}_1\boldsymbol{a}_N & \mathbf{0}^D \\ flt(\boldsymbol{a}_1\boldsymbol{a}_1^T) & \dots & flt(\boldsymbol{a}_N\boldsymbol{a}_N^T) & flt(\mathbf{0}^D(\mathbf{0}^D)^T) \end{pmatrix}$$

Layer 2:

$$\underbrace{\left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \end{array}\right)}_{conv(in\_proj(\cdot))} \odot \underbrace{\left(\begin{array}{c} \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1} \rightarrow \\ \hline \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1}^{D^2} \rightarrow \end{array}\right)}_{gate\_proj(\cdot)} = \left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \end{array}\right)$$

$$\left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \end{array}\right) \underbrace{\rightarrow}_{out\_proj=Identity} \left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \end{array}\right)$$

Now, we use a single BASECONV layer to implement a gradient descent update.

$$\underbrace{\left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \mathbf{1}^D \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \end{array}\right)}_{conv(in\_proj(\cdot))} \odot \underbrace{\left(\begin{array}{c} \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1} \rightarrow \\ \hline \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1}^D \rightarrow \\ \hline \leftarrow \mathbf{1}^{D^2} \rightarrow \\ \hline \mathbf{0}^D \dots \mathbf{0}^D \mathbf{1}^D \\ \hline \mathbf{0}^{D^2} \dots \mathbf{0}^{D^2} flt(\mathbf{1}^D \boldsymbol{x}_0^T) \end{array}\right)}_{gate\_proj(\cdot)} = \left(\begin{array}{cccc} \boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\ \hline \boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\ \hline \mathbf{0}^D & \dots & \mathbf{0}^D & \mathbf{1}^D \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\ \hline \multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \\ \hline \multicolumn{4}{c}{\mathbf{0}^D \dots \mathbf{0}^D \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i} \\ \hline \mathbf{0}^{D^2} \dots \mathbf{0}^{D^2} \sum_{i=1}^N flt(\boldsymbol{a}_i (\boldsymbol{a}_i \odot \boldsymbol{x}_0)^T) \end{array}\right)$$

Note that the gradient

$$\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}_0) = \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i - \left( \sum_{i=1}^N \boldsymbol{a}_i \boldsymbol{a}_i^T \right) \boldsymbol{w_0}$$

can be written as a linear combination of the vector

$$\left(\begin{array}{c} \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \\ \hline \sum_{i=1}^N flt(\boldsymbol{a}_i (\boldsymbol{a}_i \odot \boldsymbol{x}_0)^T) \end{array}\right)$$

so we can write a weight construction for $out\_proj$ that updates $w_0 \to w_0 - \eta \nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}_0)$:

$$\left(\begin{array}{cccc}
\boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\
\hline
\boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\
\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 \\
\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \mathbf{1}^D \\
\hline\hline
\multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\
\hline
\multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \\
\hline\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \\
\hline
\mathbf{0}^{D^2} & \dots & \mathbf{0}^{D^2} & \sum_{i=1}^N flt(\boldsymbol{a}_i (\boldsymbol{a}_i \odot \boldsymbol{x}_0)^T)
\end{array}\right)
\overset{\rightarrow}{\underset{out_proj}{}}
\left(\begin{array}{cccc}
\boldsymbol{a}_1 & \dots & \boldsymbol{a}_N & \mathbf{0}^D \\
\hline
\boldsymbol{b}_1 & \dots & \boldsymbol{b}_N & 0 \\
\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \boldsymbol{x}_0 - \eta \nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}_0) \\
\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \mathbf{1}^D \\
\hline\hline
\multicolumn{4}{c}{\leftarrow \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \rightarrow} \\
\hline
\multicolumn{4}{c}{\leftarrow \sum_{i=1}^N flt(\boldsymbol{a}_i \boldsymbol{a}_i^T) \rightarrow} \\
\hline\hline
\mathbf{0}^D & \dots & \mathbf{0}^D & \sum_{i=1}^N \boldsymbol{b}_i \boldsymbol{a}_i \\
\hline
\mathbf{0}^{D^2} & \dots & \mathbf{0}^{D^2} & \sum_{i=1}^N flt(\boldsymbol{a}_i (\boldsymbol{a}_i \odot \boldsymbol{x}_0)^T)
\end{array}\right)$$

### D.3.2 Lower bounds: BaseConv constructions are asymptotically optimal

Note that the non-causal weight construction in Appendix D.3.1 requires $O(1)$ layers and $O(D)$ state size, while the causal weight construction in Appendix D.3.1 requires $O(1)$ layers and $O(D^2)$ state size. Clearly the $O(D)$ state size requirement for non-causal models is tight, since one needs to store the gradient $\nabla_{\boldsymbol{x}} \mathcal{L} \in \mathbb{R}^D$. In this section, we prove that the $O(D^2)$ state size requirement for causal models is also asymptotically tight.

**Theorem D.33.** *Any single-pass (causal) algorithm computing the gradient*

$$\nabla_{\boldsymbol{x}} \mathcal{L} = \sum_{j=1}^N b_j \boldsymbol{a}_j - \left( \sum_{j=1}^N \boldsymbol{a}_j \boldsymbol{a}_j^T \right) \boldsymbol{x}$$

*given inputs* $\{(\boldsymbol{a}_1, b_1), \dots, (\boldsymbol{a}_N, b_N); \boldsymbol{x}\}$*, with* $(\boldsymbol{a}_i, b_i) \in \mathbb{R}^{(D+1)N}$ *and* $\boldsymbol{x} \in \mathbb{R}^D$*, requires* $\Omega(D^2)$ *state size in the worst case, where* $b_j \in \mathbb{R}$ *and* $\boldsymbol{a}_j, \boldsymbol{x} \in \mathbb{R}^D$*.*

*Proof.* For simplicity, we pick $N = D$ for large enough $D$.

Since we can compute $\sum_{j=1}^D b_j \boldsymbol{a}_j$ in $O(D)$ space, we focus on computing the expensive $\left( \sum_{j=1}^N \boldsymbol{a}_j \boldsymbol{a}_j^T \right) \boldsymbol{x}$ term. Assume there exists a single-pass algorithm $\mathcal{A}$ that computes $\left( \sum_{j=1}^N \boldsymbol{a}_j \boldsymbol{a}_j^T \right) \boldsymbol{x}$ exactly for all choices of $\boldsymbol{a}_1, \dots, \boldsymbol{a}_D, \boldsymbol{x} \in \mathbb{R}^D$. Now consider the following two claims:

1. Define $\boldsymbol{s}_D$ to be the state of the algorithm after seeing $\boldsymbol{a}_1, \dots, \boldsymbol{a}_D$. Then we claim that $\boldsymbol{s}_D$ must have enough information to exactly reconstruct $\boldsymbol{M}_D := \sum_{j=1}^D \boldsymbol{a}_j \boldsymbol{a}_j^T$.

   This follows since the algorithm must be correct for any value $\boldsymbol{x} \in \mathbb{R}^D$ takes on. In particular, setting $\boldsymbol{x} = \boldsymbol{e}_i$ for $i \in [D]$, we observe that the algorithm must be able to exactly recover $\boldsymbol{M}_D \boldsymbol{e}_i = \boldsymbol{M}_D[:, i]$, $i \in [D]$.

2. The space of matrices

   $$\left\{ \sum_{j=1}^D \boldsymbol{a}_j \boldsymbol{a}_j^T \right\}$$

   over all choices of $\boldsymbol{a}_j \in \mathbb{R}^D$, $j \in [d]$ contains the set of all real symmetric matrices in $\mathbb{R}^{D \times D}$.

This holds since for any real symmetric matrix $\boldsymbol{A}$, we can obtain a set of possible $\boldsymbol{a}_j$'s via its eigendecomposition Strang (2012):

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T = \sum_{j=1}^{D} \boldsymbol{a}_j \boldsymbol{a}_j^T$$

where $\boldsymbol{a}_j = \sqrt{\lambda_j}\boldsymbol{Q}[:, j]$.

From the first claim, we conclude that $\boldsymbol{s}_D$ must contain enough information to be able to recover $\boldsymbol{M}_D$ for any possible value $\boldsymbol{M}_D$ can take on (over all choices of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_D \in \mathbb{R}^D$). From the second claim, we have that the space of possible values of $\boldsymbol{M}_D$ includes the set of all possible real symmetric matrices. Since we know that this set requires $\frac{(D)(D+1)}{2}$ parameters to represent, we can conclude that $|\boldsymbol{s}_D| \geq \frac{(D)(D+1)}{2} \geq \Omega(D^2)$. □

### D.4 BASECONV AND JACKSON'S THEOREM

In this section we prove BASECONV's ability to approximate arbitrary univariate and multivariate smooth functions.

We start with a special case of smooth functions that apply entry-wise univariate smooth functions:

**Definition D.34.** Let $\overline{f} : [-1, 1] \to \mathbb{R}$ be a $(k, L)$-smooth univariate function. Then define

$$f : [-1, 1]^{N \times D} \to \mathbb{R}^{N \times D}$$

as follows. For all $0 \leq i < N$, $0 \leq j < D$, and $\mathbf{u} \in [-1, 1]^{N \times D}$:

$$(f(\mathbf{u}))[i, j] = \overline{f}(\mathbf{u}[i, j]).$$

Now we will state a simple observation on BASECONV's ability to approximate these functions.

**Lemma D.35.** *For any smooth function $f$ as defined in Definition D.34, let $g(\mathbf{x}) = P_{\overline{f}}(\mathbf{x})$ with $P_{\overline{f}}$ being the polynomial from Corollary D.12. Then for all $\mathbf{x} \in [-1, 1]^{N \times D}$,*

$$\|g(\mathbf{x}) - f(\mathbf{x})\|_\infty \leq \epsilon.$$

*Proof.* Follows from Definitions D.7 and D.34 and Corollary D.12. □

Next we will state a construction of an arithmetic circuit for a function that applies a univariate polynomial to all entries in $[-1, 1]^{N \times D}$:

**Lemma D.36.** *Let $P(X)$ be a degree $d$ univariate polynomial. Then there is a $(ND, O(ND), O(d), ND)$-*circuit *to compute $P(\mathbf{u})$ where $P(\mathbf{u})$ is defined as follows. For an input $\mathbf{u} \in [-1, 1]^{N \times D}$,*

$$P(\mathbf{u})[i, j] = P(\mathbf{u}[i, j]).$$

*Proof.* Let the univariate polynomial be

$$P(X) = \sum_{i=0}^{d} c_i X^i$$

where coefficients $c_i \in \mathbb{R}$.

Next we state the natural arithmetic circuit to compute $P(x)$ for $x \in \mathbb{R}$ in Algorithm 2:

**Algorithm 2** circuit $\mathcal{C}_P(x)$:

$s_0 \leftarrow c_0$
$m_0 \leftarrow 1$
**for** $j = 1, 2, \ldots, d$ **do**
    $m_j \leftarrow m_{j-1} \cdot x$ ▷ Multiplication gate
    $t_j \leftarrow c_j \cdot m_j$ ▷ Multiplication gate
    $s_j \leftarrow s_{j-1} + t_j$ ▷ Addition gate
**return** $s_d$ ▷ $s_d$ is the output gate

Next we apply the above circuit in parallel to form the circuit that computes $P(\mathbf{u})$ in Algorithm 3:

**Algorithm 3** Circuit for $P(\mathbf{u})$:

**for** $i = 0, 1, \ldots, N - 1$ **do**
    **for** $j = 0, 1, \ldots, D - 1$ **do**
        $\mathbf{z}[i, j] = \mathcal{C}_P(\mathbf{u}[i, j])$ ▷ Do this in parallel
**return** $\mathbf{z}$ ▷ $\mathbf{z}$ is the output matrix

Looking at Algorithm 2, the depth of the circuit is $3d$, or $O(d)$, since that is the bound on iterations of the for loop, and each iteration we compute 3 sequential operations. Therefore it's a $(1, O(d), O(d, O(1))$-*circuit*.

For Algorithm 3, The width is $O(ND)$, since we have our input of size $N \times D$, which goes through the circuit in parallel, as stated in Algorithm 3. Therefore we have an $(ND, O(ND), O(d), O(ND))$-*circuit* that computes $P(\mathbf{u})$. □

Since BASECONV has the ability to represent any arithmetic circuit, we get the following:

**Corollary D.37.** *We can implement $P(\mathbf{u})$ (where $P(\mathbf{u})$ is as defined in Lemma D.36 ) when $\deg(P) = d$ with a $(N, O(d \log(ND)), D, O(ND), D)$ − BASECONV.*

*Proof.* Follows from Lemma D.36 giving us the $(ND, O(ND), O(d), O(ND))$-*circuit* for an arbitrary polynomial and Theorem D.30 gives us the BASECONV model to implement the circuit. □

We will prove a tighter bound showing we can represent $P(\mathbf{u})$ using a constant number of BASECONV layers (for constant $\deg(P)$):

**Theorem D.38.** *We can implement $P(\mathbf{u})$ when $\deg(P) = d$ with an $(O(N), O(d), D, O(N), D)$ − BASECONV model.*

*Proof.* We will convert the steps done in Algorithm 2 to layers of BASECONV. Since Algorithm 3 is essentially running Algorithm 2 in parallel over all entries of input $\mathbf{u} \in [-1, 1]^{N \times D}$, the latter happens automatically in our BASECONV implementation.

For this proof, define

$$P_j(X) = X^j$$

and let $\boldsymbol{C}_i$ be the matrix of size $N \times D$ and all the entries are $c_i$.

We expand the input to our BASECONV layers as follows,

$$\mathbf{u} = \begin{pmatrix} \mathbf{u}' \\ \mathbf{0}^{3N \times D} \end{pmatrix}.$$

This means that the size of the internal dimension of our BASECONV layers will be $(4N, D)$.

To begin iterations of the for loop we need to store initial values into the extra space in $\mathbf{u}$. Taking us from

$$\mathbf{u} = \begin{pmatrix} \mathbf{u}' \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix} \to \begin{pmatrix} \mathbf{u} \\ \mathbf{1}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \boldsymbol{C}_0 \end{pmatrix} =: \mathbf{u}_0$$

We do this via BASECONV$(\mathbf{u}', \boldsymbol{I}^{D \times D}, \begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \boldsymbol{C}_0 \end{pmatrix}, \mathbf{0}^{4N \times D}, \mathbf{1}^{4N \times D})$ which computes

$$\left( \begin{pmatrix} \mathbf{u} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix} \boldsymbol{I}^{D \times D} + \begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \boldsymbol{C}_0 \end{pmatrix} \right) \odot \left( \mathbf{0}^{4N \times D} * \begin{pmatrix} \mathbf{u} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix} + \mathbf{1}^{4N \times D} \right).$$

The above simplifies to

$$\left( \begin{pmatrix} \mathbf{u} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix} + \begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \boldsymbol{C}_0 \end{pmatrix} \right) \odot \left( \mathbf{1}^{4N \times D} \right),$$

which gives us

$$\begin{pmatrix} \mathbf{u} \\ \mathbf{1}^{N \times D} \\ \mathbf{1}^{N \times D} \\ \boldsymbol{C}_0 \end{pmatrix} =: \mathbf{u}_0,$$

as desired

This was done with a $(4N, 1, D, 4N, D) -$ BASECONV layer.

Our goal is, at the end of iteration $j$ to compute $\mathbf{u}_j \in \mathbb{R}^{4N \times D}$ such that,

$$\mathbf{u}_j = \left(\begin{array}{c} \mathbf{u} \\ \hline P_j(\mathbf{u}) \\ \hline \boldsymbol{C}_j \odot P_j(\mathbf{u}) \\ \hline \boldsymbol{C}_0 + \boldsymbol{C}_1 \odot P_1(\mathbf{u}) + \cdots + \boldsymbol{C}_j \odot P_j(\mathbf{u}) \end{array}\right).$$

We will view the above matrix in terms of the variables in the Algorithm 2 as follows

$$\left(\begin{array}{c} \mathbf{u} \\ \hline P_j(\mathbf{u}) \\ \hline \boldsymbol{C}_j \odot P_j(\mathbf{u}) \\ \hline \boldsymbol{C}_0 + \boldsymbol{C}_1 \odot P_1(\mathbf{u}) + \cdots + \boldsymbol{C}_j \odot \mathbf{u}^j \end{array}\right) =: \left(\begin{array}{c} \mathbf{u} \\ \hline \mathbf{m}_j \\ \hline \mathbf{t}_j \\ \hline \mathbf{s}_j \end{array}\right).$$

The for loop runs for values of $1 \leq j \leq d$ which the remainder of this proof will replicate. There are three lines in the for loop in Algorithm 2 which we will cover how these operations happen in constant number of BASECONV layers.

In line 4, the first line in the for loop computes

$$\mathbf{u}_{j-1} = \left(\begin{array}{c} \mathbf{u} \\ \hline \mathbf{m}_{j-1} \\ \hline \mathbf{t}_{j-1} \\ \hline \mathbf{s}_{j-1} \end{array}\right) \to \left(\begin{array}{c} \mathbf{u} \\ \hline \mathbf{m}_{j} \\ \hline \mathbf{t}_{j-1} \\ \hline \mathbf{s}_{j-1} \end{array}\right) =: \mathbf{u}_j^{(1)}.$$

Note that $\mathbf{m}_j = \mathbf{m}_{j-1} \odot \mathbf{u}$.

We use the `remember` primitive to compute $\mathbf{u}_j^{(1)}$ from $\mathbf{u}_{j-1}$. Define $f : \mathbb{R}^{2N \times D} \to \mathbb{R}^{2N \times D}$ as follows

$$f\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} = \begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \odot \mathbf{u} \end{pmatrix}.$$

If we can compute $f$ with BASECONV layers then we can compute $\mathbf{u}_j^{(1)}$ for $\mathbf{u}_{j-1}$ by calling $\texttt{remember}(\mathbf{u}_j, 0, 2N-1, f)$.

We show BASECONV $\left(\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_j \end{pmatrix}, \boldsymbol{I}^{D \times D}, \mathbf{0}^{2N \times D}, \boldsymbol{H}, \begin{pmatrix} \mathbf{1}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix}\right)$ maps

$$\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} \to \begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j} \end{pmatrix},$$

where $\boldsymbol{H}$ is defined as in Proposition D.24. We plug the matrices into the BASECONV layer as follows:

$$\left(\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} \cdot \boldsymbol{I}^{D \times D} + \mathbf{0}^{2N \times D}\right) \odot \left(\boldsymbol{H} * \begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix}\right).$$

We know from Proposition D.24 that this convolution operation is a shift down by $N$ rows. Therefore the above simplifies to

$$\left(\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} \cdot \boldsymbol{I}^{D \times D} + \mathbf{0}^{2N \times D}\right) \odot \left(\begin{pmatrix} \mathbf{0}^{N \times D} \\ \mathbf{u} \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N \times D} \\ \mathbf{0}^{N \times D} \end{pmatrix}\right),$$

which simplifies to

$$\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \end{pmatrix} \odot \begin{pmatrix} \mathbf{1}^{N \times D} \\ \mathbf{u} \end{pmatrix} = \begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j-1} \odot \mathbf{u} \end{pmatrix} = f\begin{pmatrix} \mathbf{u} \\ \mathbf{m}_{j} \end{pmatrix},$$

as desired. Therefore by Proposition D.26, line 4 can be computed by $(4N, 8, D, 4N, D) -$ BASECONV.

For line 5 of the for loop we need to compute

$$\mathbf{u}_j^{(1)} = \begin{pmatrix} \mathbf{u} \\ \hline \mathbf{m}_j \\ \hline \mathbf{t}_{j-1} \\ \hline \mathbf{s}_{j-1} \end{pmatrix} \to \begin{pmatrix} \mathbf{u} \\ \hline \mathbf{m}_j \\ \hline \mathbf{t}_{j} \\ \hline \mathbf{s}_{j-1} \end{pmatrix} =: \mathbf{u}_j^{(2)}.$$

Note that $\mathbf{t}_j = \boldsymbol{C}_j \odot \mathbf{m}_j$.

To do this we will use three BASECONV layers. We use the `remember` primitive to compute $\mathbf{u}_j^{(2)}$ from $\mathbf{u}_j^{(1)}$. Define $g : \mathbb{R}^{2N\times D} \to \mathbb{R}^{2N\times D}$ as follows,

$$g\begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_j \\ \boldsymbol{C}_j \odot \mathbf{m}_j \end{pmatrix}.$$

If we can compute $g$ with BASECONV layers then we can compute $\mathbf{u}_j^{(2)}$ for $\mathbf{u}_{j-1}$ by calling $\texttt{remember}(\mathbf{u}_j^{(1)}, N, 3N-1, g)$.

Indeed, we show the $g$ can be computed by first computing $\text{BASECONV}\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix}, \boldsymbol{I}^{D\times D}, \mathbf{0}^{2N\times D}, \mathbf{0}^{2N\times D}, \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right)$:

$$\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix} \cdot \mathbf{I}^{D\times D} + \mathbf{0}^{2N\times D}\right) \odot \left(\mathbf{0}^{2N\times D} * \begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right),$$

which simplifies to

$$\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix}\right) \odot \left(\begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right).$$

This results in

$$\begin{pmatrix} \mathbf{m}_j \\ \mathbf{0}^{N\times D} \end{pmatrix}.$$

We pass into the next layer, $\text{BASECONV}\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{0}^{N\times D} \end{pmatrix}, \boldsymbol{I}^{D\times D}, \begin{pmatrix} \mathbf{0}^{N\times D} \\ \mathbf{1}^{N\times D} \end{pmatrix}, \boldsymbol{H}, \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right)$ where $\boldsymbol{H}$ is defined as in Proposition D.24:

$$\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{0}^{N\times D} \end{pmatrix} \cdot \boldsymbol{I}^{D\times D} + \begin{pmatrix} \mathbf{0}^{N\times D} \\ \mathbf{1}^{N\times D} \end{pmatrix}\right) \odot \left(\boldsymbol{H} * \begin{pmatrix} \mathbf{m}_j \\ \mathbf{0}^{N\times D} \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right).$$

Since the kernel $\boldsymbol{H}$ is as in Proposition D.24, this simplifies to

$$\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{1}^{N\times D} \end{pmatrix} \odot \left(\begin{pmatrix} \mathbf{0}^{N\times D} \\ \mathbf{m}_j \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{0}^{N\times D} \end{pmatrix}\right)\right).$$

The above simplifies further to

$$\begin{pmatrix} \mathbf{m}_j \\ \mathbf{1}^{N\times D} \end{pmatrix} \odot \begin{pmatrix} \mathbf{1}^{N\times D} \\ \mathbf{m}_j \end{pmatrix},$$

which results in:

$$\begin{pmatrix} \mathbf{m}_j \\ \mathbf{m}_j \end{pmatrix}.$$

We pass the above to $\text{BASECONV}\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{m}_j \end{pmatrix}, \boldsymbol{I}^{D\times D}, \mathbf{0}^{2N\times D}, \mathbf{0}^{2N\times D}, \begin{pmatrix} \mathbf{1}^{N\times D} \\ \boldsymbol{C}_j \end{pmatrix}\right)$:

$$\left(\begin{pmatrix} \mathbf{m}_j \\ \mathbf{m}_j \end{pmatrix} \cdot \boldsymbol{I}^{D\times D} + \mathbf{0}^{2N\times D}\right) \odot \left(\mathbf{0}^{2N\times D} * \begin{pmatrix} \mathbf{m}_j \\ \mathbf{m}_j \end{pmatrix} + \begin{pmatrix} \mathbf{1}^{N\times D} \\ \boldsymbol{C}_j \end{pmatrix}\right)$$

which simplifies to

$$\begin{pmatrix} \mathbf{m}_j \\ \mathbf{m}_j \end{pmatrix} \odot \begin{pmatrix} \mathbf{1}^{N\times D} \\ \boldsymbol{C}_j \end{pmatrix}.$$

The above results in

$$\begin{pmatrix} \mathbf{m}_j \\ \boldsymbol{C}_j \odot \mathbf{m}_j \end{pmatrix} = g\begin{pmatrix} \mathbf{m}_j \\ \mathbf{t}_{j-1} \end{pmatrix},$$

as desired.

Therefore by Corollary D.27, line 5 was computed by $(4N, O(1), D, 4N, D) - \text{BASECONV}$.

For line 6, the final line of the for loop, we want

$$\mathbf{u}_j^{(2)} = \left(\begin{array}{c}\mathbf{u}\\\hline \mathbf{m}_j\\\hline \mathbf{t}_j\\\hline \mathbf{s}_{j-1}\end{array}\right) \to \left(\begin{array}{c}\mathbf{u}\\\hline \mathbf{m}_j\\\hline \mathbf{t}_j\\\hline \mathbf{s}_j\end{array}\right) =: \mathbf{u}_j.$$

Note that $\mathbf{s}_j = \mathbf{s}_{j-1} + \mathbf{t}_j$

Define function $h : \mathbb{R}^{2N\times D} \to \mathbb{R}^{2N\times D}$ as follows,

$$h\begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix} = \begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1} + \mathbf{t}_j\end{pmatrix}.$$

If we can compute $h$ with BASECONV layers then we can compute $\mathbf{u}_j$ for $\mathbf{u}_{j-1}$ by calling $\texttt{remember}(\mathbf{u}_j^{(2)}, 2N, 4N-1, h)$.

Indeed we show that $h$ can be computed by computing $\text{BASECONV}\left(\begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix}, \mathbf{0}^{D\times D}, \mathbf{1}^{2N\times D}, \overline{\boldsymbol{H}}, \mathbf{0}^{2N\times D}\right)$, where kernel $\overline{\boldsymbol{H}} \in \mathbb{R}^{2N\times D}$ is defined as:

$$\overline{\boldsymbol{H}}[k,:] \equiv \begin{cases}\mathbf{1}^D & \text{if } k \in \{0, N\}\\ \mathbf{0}^D & \text{otherwise.}\end{cases}$$

.

This layer computes

$$\left(\begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix} \cdot \mathbf{0}^{2N\times D} + \mathbf{1}^{2N\times D}\right) \odot \left(\overline{\boldsymbol{H}} * \begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix} + \mathbf{0}^{2N\times D}\right).$$

This simplifies to

$$\left(\mathbf{1}^{2N\times D}\right) \odot \left(\overline{\boldsymbol{H}} * \begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix}\right) = \left(\overline{\boldsymbol{H}} * \begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix}\right).$$

Now we compute this convolution for column $i$, $0 \le i < 2N$. For notational convenience, let $\begin{pmatrix}\mathbf{t}_j\\ \mathbf{s}_{j-1}\end{pmatrix}$ be noted as matrix $\mathbf{V}$. Then we have:

$$\overline{\boldsymbol{H}}[:,i] * \mathbf{V}[:,i] = \text{coeff}\left((1+X^N)\mathbf{V}[:,i](X) \mod X^{2N}\right),$$

where $(1+X^N)$ is the polynomial representation of the columns of $\overline{\boldsymbol{H}}$ (since there's a one in the 0th index and a one in the $N$th index of each column).

The expression simplifies to

$$\text{coeff}\mathbf{V}[:,i](X) + \mathbf{V}[:,i](X)X^N \mod X^{2N},$$

which can be broken down to

$$\begin{aligned}&\text{coeff}\left(\left(\mathbf{V}[0][i] + \mathbf{V}[1][i]X + \cdots + \mathbf{V}[2N-1][i]X^{2N-1}\right) \mod X^{2N}\right)\\ &+ \text{coeff}\left(\left(\mathbf{V}[0][i]X^N + \mathbf{V}[1][i]X^{N+1} + \cdots + \mathbf{V}[2N-1][i]X^{3N-1}\right) \mod X^{2N}\right)\end{aligned}$$

with the lower order terms in the second coefficient vector being zeros,

$$\begin{aligned}&\text{coeff}\left(\left(\mathbf{V}[0][i] + \mathbf{V}[1][i]X + \cdots + \mathbf{V}[2N-1][i]X^{2N-1}\right) \mod X^{2N}\right)\\ &+ \text{coeff}\left(\left(0 + 0X + \cdots + 0X^{N-1} + \mathbf{V}[0][i]X^N + \cdots + \mathbf{V}[2N-1][i]X^{3N-1}\right) \mod X^{2N}\right)\end{aligned}$$

After taking $\mod X^{2N}$ we get

$$\begin{aligned}&\text{coeff}\left(\mathbf{V}[0][i] + \mathbf{V}[1][i]X + \cdots + \mathbf{V}[2N-1][i]X^{2N-1}\right)\\ &+ \text{coeff}\left(0 + 0X + \cdots 0X^{N-1}\mathbf{V}[0][i]X^N + \cdots \mathbf{V}[N-1][i]X^{2N-1}\right)\end{aligned}$$

The first set of coefficients is the input matrix as is. And the second one is the input matrix shifted down as seen in Proposition D.24. Therefore when we add these vectors we are doing

$$\begin{pmatrix} \mathbf{t}_j \\ \mathbf{s}_{j-1} \end{pmatrix} + \begin{pmatrix} \mathbf{0}^{N\times D} \\ \mathbf{t}_j \end{pmatrix} = h \begin{pmatrix} \mathbf{t}_j \\ \mathbf{s}_{j-1} \end{pmatrix},$$

as desired. Therefore by Proposition D.26, line 6 is computed with by $(4N, 1, D, 4N, D)$ − BASECONV.

The $\mathbf{s}_d$ matrix gives us $\boldsymbol{C}_0 + \boldsymbol{C}_1 \odot \mathbf{m}_1 + \cdots + \boldsymbol{C}_d \odot \mathbf{m}_d$. Recalling that

$$\boldsymbol{C}_0 + \boldsymbol{C}_1 \odot \mathbf{m}_1 + \cdots + \boldsymbol{C}_d \odot \mathbf{m}_d \equiv \sum_{j=0}^{d} \boldsymbol{C}_j \odot \mathbf{u}^j = P(\mathbf{u}),$$

and hence $\mathbf{s}_d$ is our desired output.

We have $d$ layers, each consisting of $O(1)$ BASECONV layers. Giving us $O(d)$ many layers to implement Algorithm 2.

Therefore, via the ability to stack BASECONV layers to do function composition, the for loop was computed by a $(4N, O(d), D, 4N, D)$ − BASECONV , as desired. □

The following states BASECONV's ability to approximate a univariate smooth function:

**Proposition D.39.** *Let $f$ be the $(k, L)$ -smooth function defined in Definition D.34. Then there is a $\left(N, O\left(\sqrt[k]{\frac{L}{\epsilon}}\right) + k, D, (ND), D\right)$ − BASECONV model that approximates $f$ within error $\epsilon$.*

*Proof.* Follows from Corollary D.12, Lemma D.35, and Theorem D.38. □

### D.5 MULTIVARIATE FUNCTION APPROXIMATION

We begin by defining more multivariate notation.

We consider the following multivariate functions:

**Definition D.40.** For $0 \leq 1 < N, 0 \leq j < D$, let $\bar{f}_{i,j} : [-1, 1]^{N\times D} \to \mathbb{R}$ be a $(k, L)$-smooth multivariate function. Then define

$$f(\mathbf{x}) : [-1, 1]^{N\times D} \to \mathbb{R}^{N\times D}$$

as follows. For all $0 \leq i < N, 0 \leq j < D$, $\mathbf{u} \in [-1, 1]^{N\times D}$ define

$$f(\mathbf{u})[i, j] := \bar{f}_{i,j}(\mathbf{u}).$$

**Lemma D.41.** *For any smooth function $f$ as defined in Definition D.40, let $g(X_1, \ldots, X_{N\times D}) = P_{\bar{f}}(X_1, \ldots, X_{N\times D})$ be the polynomial from Corollary D.14. Then for all $\mathbf{x} \in [-1, 1]^{N\times D}$,*

$$\|g(\mathbf{x}) - f(\mathbf{x})\|_\infty \leq \epsilon.$$

*Proof.* Follows from Definitions D.7 and D.40 and Corollary D.14. □

Next we will state a construction for an arithmetic circuit for a function that takes a $[-1, 1]^{N\times D}$ variable input:

**Lemma D.42.** *Let $P(\boldsymbol{X})$ be a degree $d$ multivariate polynomial. Then there is a $\left(n, O(d \cdot n^d), O(d \log(n)), O(n^d)\right)$*-circuit *to compute $P(\mathbf{u})$ on any input $\mathbf{u} \in [-1, 1]^n$.*

*Proof.* Let the multivariate polynomial be as defined in Definition D.6. We build the circuit to compute this in Algorithm 4,

**Algorithm 4** circuit $\mathcal{C}_P(\mathbf{x})$:

1: **for** $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}^n_{\geq 0}$ such that $\sum_{i=1}^n \alpha_i \leq d$ **do**
2: $\quad m_{\boldsymbol{\alpha}} \leftarrow 1$
3: $\quad$ **for** $i = 1, 2, \ldots, n$ **do** ▷ Done in parallel
4: $\quad\quad$ **if** $\alpha_i \neq 0$ **then**
5: $\quad\quad\quad m_{\boldsymbol{\alpha}} \leftarrow m_{\boldsymbol{\alpha}} \cdot x_i^{\alpha_i}$
6: $\quad t_{\boldsymbol{\alpha}} \leftarrow c_{\boldsymbol{\alpha}} \cdot m_{\boldsymbol{\alpha}}$
7: **for** $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}^n_{\geq 0}$ such that $\sum_{i=1}^n \alpha_i \leq d$ **do**
8: $\quad s \leftarrow \sum t_{\boldsymbol{\alpha}}$ ▷ Done in parallel
9: **return** s

We compute the for loop starting on line 3 by making multiplications in parallel. Therefore obtaining a depth of $O(\log(d))$. We also have the for loop starting on line 7, making pairwise addition operations, resulting in a depth of $O(d \log(n))$. □

We again use the result that BASECONV can represent any arithmetic circuit to get:

**Corollary D.43.** *We can implement $P(\mathbf{u})$ (where $P(\mathbf{u})$ is as defined in Lemma D.42) when $\deg(P(X_1, \ldots, X_{ND})) = d$ with a $\big(N, O(d \log(ND)), D, O((ND)^d), D\big) -$ BASECONV where $\mathbf{u} \in [-1, 1]^{N \times D}$.*

*Proof.* Lemma D.42 gives us the arithetmic circuit that computes this polynomial. Then via Theorem D.30 we get a $\big(N, O(d \log(ND)), D, O((ND)^d), D\big) -$ BASECONV model to implement the circuit. □

Finally we state BASECONV's ability to approximate multivariate smooth functions:

**Proposition D.44.** *Let $f$ be the function defined in Definition D.40. Then there is a $\big(N, O(d \log(ND)), D, O((ND)^d), D\big) -$ BASECONV model that approximates $f$ to within error $\epsilon$, with $d = O_k(\sqrt[k]{\frac{NDL}{\epsilon}})$.*

*Proof.* We get the existence of a polynomial that approximates $f$ for some $\epsilon$ from Corollary D.14. Then via Corollary D.43 we get that we can represent any polynomial, implying $\big(N, O(d \log(ND)), D, O((ND)^d), D\big) -$ BASECONV represents any polynomial that approximates the multivariate smooth function $f$. □

### D.6 NOTATION FOR D.7

This notation section is for the succeeding subsection which will prove we can recover the functions Square and Linear exactly given some assumptions and expected gradients of their respective loss functions being 0.

When indexing an entry of a 2 dimensional matrix, $\boldsymbol{A}$ we denote it as $\boldsymbol{A}_{i,j}$ where $i$ is the row number and $j$ is the column.

The following are the parameters of a BASECONV layer.

1. We will use 0 indexing
2. $[N] = \{0, 1, \ldots, N-1\}$
3. $\boldsymbol{W} = \{\boldsymbol{W}_{i,j}\} \in \mathbb{R}^{d \times d}$
4. $\boldsymbol{K} = \{\boldsymbol{K}_{i,j}\} \in \mathbb{R}^{N \times d}$
5. $\boldsymbol{B}^{(1)} = \{\boldsymbol{B}^{(1)}_{i,j}\} \in \mathbb{R}^{N \times d}$
6. $\boldsymbol{B}^{(2)} = \{\boldsymbol{B}^{(2)}_{i,j}\} \in \mathbb{R}^{N \times d}$
7. $\boldsymbol{\theta} = \left(\boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(1)}, \boldsymbol{B}^{(2)}\right)$
8. $\mathbf{u}_{i:} \stackrel{\text{def}}{=} \mathbf{u}[i, :]$
9. Input to a BASECONV layer, $\mathbf{u} = \{\mathbf{u}_{i,j}\} \in \mathbb{R}^{N \times d}$
10. $\mathbb{E}\left[\{\boldsymbol{M}_{i,j}\}\right] = \{\mathbb{E}\left[\boldsymbol{M}_{i,j}\right]\}$
11. BASECONV layer operation,

$$\boldsymbol{Z} = \text{BASECONV}\left(\boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(1)}, \boldsymbol{B}^{(2)}, \mathbf{u}\right) \stackrel{\text{def}}{=} \left(\mathbf{u}\boldsymbol{W} + \boldsymbol{B}^{(1)}\right) \odot \left(\boldsymbol{K} * \mathbf{u} + \boldsymbol{B}^{(2)}\right) \quad (48)$$

12. Our target function:

$$f : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d}$$

We'll denote $\left(f(\mathbf{u})\right)[i, j]$ by $f(\mathbf{u})_{i,j}$.

We need to define the training input distribution to begin talking about expected values of the layers.

#### D.6.1 TRAINING INPUT DISTRIBUTION

1. Let $\Delta$ be the training distribution on $\mathbb{R}^{N \times d}$

**Assumption D.45.** Given a monomial $\mathbb{E}\left[\Pi_e \left(\mathbf{u}_{ie,je}\right)^{me}\right] = 0$ if $me$ is odd. Otherwise, $\mathbb{E}\left[\Pi_e \left(\mathbf{u}_{ie,je}\right)^{me}\right] > 0$.

**Assumption D.46.** Assume that the training data is generated as

- $\mathbf{u} \sim \Delta$ as input
- Output is $\mathbf{y} = f(\mathbf{u}) + \mathcal{E}$ where $\mathcal{E} = \{\mathcal{E}_{i,j}\} \in \mathbb{R}^{N \times d}$ is the random error matrix such that
  - The distribution on $\mathcal{E}$ and $\Delta$ are independent. (Call the distribution on $\mathcal{E}$ to be $\Delta_{\mathcal{E}}$)
  - $\mathbb{E}[\mathcal{E}_{i,j}] = 0$ for all $(i, j) \in [N] \times [d]$

**Loss function**

- Define
$$\overline{L_{ij}}(\mathbf{u}, \boldsymbol{\theta}, \mathcal{E}) = (z_{ij} - y_{ij})^2 = (z_{ij} - f(\mathbf{u})_{ij} - \mathcal{E}_{ij})^2 \tag{49}$$
- $L(\mathbf{u}) = \sum_{i,j} \overline{L_{ij}}(\mathbf{u}, \boldsymbol{\theta}, \mathcal{E})$
- Training loss, $\overline{L^{(t)}}(\boldsymbol{\theta}) = \overline{L^{(t)}} = \mathbb{E}_{\substack{\mathbf{u}\sim\Delta \\ \mathcal{E}\sim\Delta_\mathcal{E}}} [L(\mathbf{u})]$
- $\nabla_{\boldsymbol{\theta}} \overline{L^{(t)}}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{u},\mathcal{E}} \sum_{i,j} \nabla_{\boldsymbol{\theta}} \overline{L}_{ij}(\mathbf{u})$

**The Goal** Given a target function $f$, what can we infer for $\boldsymbol{\theta} = \left(\boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(1)}, \boldsymbol{B}^{(2)}\right)$ from $\nabla_{\boldsymbol{\theta}} \overline{L}(\boldsymbol{\theta}) = \mathbf{0}$

1. Ideally, we'd like to assume that $f$ can be represented exactly by $1-$layer BASECONV.
2. For now, lets assume that $f(\mathbf{u})_{ij}$ only depends on $\mathbf{u}_{i:}$

   This includes as special cases:

   - $f(\mathbf{u})_{ij} = \mathbf{u} \odot \mathbf{u}$
   - $f(\mathbf{u}) = \mathbf{u} \cdot \overline{\boldsymbol{W}}$ for $\overline{\boldsymbol{W}} \in \mathbb{R}^{d\times d}$

We want to prove that there is a unique set of parameters that perform exactly these functions, resulting in the gradients of their loss function to be 0.

### D.6.2 A GENERIC PARTIAL DERIVATIVE

Lets try and reason as much as we can for a generic partial derivative. Let $x \in \boldsymbol{\theta} = \left(\boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(1)}, \boldsymbol{B}^{(2)}\right)$. Then from Equation (49), we have that for any $i, j$:

$$\begin{aligned}\frac{\partial \overline{L}_{i,j}}{\partial x} =& 2\left(z_{ij} - f(\mathbf{u})_{ij} - \mathcal{E}_{ij}\right)\frac{\partial Z_{i,j}}{\partial x} \\ =& 2\left(\boldsymbol{T}^{(1)}_{ij} - \boldsymbol{T}^{(2)}_{ij} - \boldsymbol{T}^{(3)}_{ij}\right),\end{aligned}$$

where

$$\begin{aligned}\boldsymbol{T}^{(1)}_{ij} &= Z_{i,j}\frac{\partial Z_{i,j}}{\partial x}. \\ \boldsymbol{T}^{(2)}_{ij} &= f(u)_{ij}\frac{\partial Z_{i,j}}{\partial x}. \\ \boldsymbol{T}^{(3)}_{ij} &= \mathcal{E}_{ij}\frac{\partial Z_{i,j}}{\partial x}.\end{aligned}$$

**Proposition D.47.** $\mathbb{E}_{\mathbf{u},\mathcal{E}}[\boldsymbol{T}^{(3)}_{ij}] = 0.$

*Proof.* $\Delta$ and $\Delta_\mathcal{E}$ are independent and $\mathbb{E}[\mathcal{E}_{ij}] = 0$ □

From now on, we'll ignore the $\boldsymbol{T}^{(3)}_{ij}$ term because of Proposition D.47 we can (in expectation) assume that $\mathcal{E} = 0$.

### D.7 SETTING THE GRADIENTS TO 0

In this section we will prove the results of BASECONV ability to exactly compute SQUARE and LINEAR given assumptions on the input data and parameters. The results are as follows. First for the SQUARE function we have

**Theorem D.48.** *Given Assumptions D.45, D.46, D.55, and a function*

$$f(\mathbf{u}) = \mathbf{u} \odot \mathbf{u}.$$

*Let* $\boldsymbol{\theta}$ *be such that,* $\mathbb{E}\nabla_{\boldsymbol{\theta}}\overline{L} = \mathbf{0}$ *then* $\textsc{BaseConv}(\mathbf{u}, \boldsymbol{\theta}) = f(\mathbf{u})$.

And the following result for the LINEAR function,

**Theorem D.49.** *Given Assumptions D.45, D.46, D.55, D.61, and a function*

$$f(\mathbf{u}) = \mathbf{u}\overline{\boldsymbol{W}}.$$

*Let* $\boldsymbol{\theta}$ *be such that,* $\mathbb{E}\nabla_{\boldsymbol{\theta}}\overline{L} = \mathbf{0}$ *then* $\textsc{BaseConv}(\mathbf{u}, \boldsymbol{\theta}) = f(\mathbf{u})$.

The following is to provide information about each entry in the output of a BASECONV layer.

**Lemma D.50.** *For all* $(i, j) \in [N] \times [d]$*, the entries of a resulting layer of* BASECONV*,* $\boldsymbol{Z}$*, are:*

$$\boldsymbol{Z}_{i,j} = \left(\left(\sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j}\right) + \boldsymbol{B}^{(1)}_{i,j}\right) \cdot \left(\left(\sum_{k=0}^{i} \boldsymbol{K}_{k,j} \cdot \mathbf{u}_{i-k,j}\right) + \boldsymbol{B}^{(2)}_{i,j}\right) \tag{50}$$

*Proof.* To begin, from Equation (48), we know that a layer of BASECONV yields a matrix $\boldsymbol{Z}$ as,

$$\boldsymbol{Z} = \left(\mathbf{u} \cdot \boldsymbol{W} + \boldsymbol{B}^{(1)}\right) \odot \left(\left(\boldsymbol{K} * \mathbf{u} + \boldsymbol{B}^{(2)}\right)\right).$$

Looking at the $\mathbf{u} \cdot \boldsymbol{W}$ operation, we know that for a row $i \in [N]$ and column $j \in [d]$, the vector dot product is computed as

$$\langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle = \sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j}.$$

Meaning each entry in the resulting matrix is defined as such

$$(\mathbf{u} \cdot \boldsymbol{W})_{i,j} = \sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j}.$$

To sum the matrix $\boldsymbol{B}^{(1)}$ to this operation, we simply add the corresponding index giving us

$$\left(\mathbf{u} \cdot \boldsymbol{W} + \boldsymbol{B}^{(1)}\right)_{i,j} = \left(\sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j}\right) + \boldsymbol{B}^{(1)}_{i,j}. \tag{51}$$

Then, for the convolution operation between $\boldsymbol{K}$ and $\mathbf{u}$, that's computed column by column, we have for all $j \in [d]$:

$$(\boldsymbol{K} * \mathbf{u})_{:,j} = \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j},$$

i.e. for any $i \in [N]$,

$$\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}\right)[i] = \sum_{k=0}^{i} \boldsymbol{K}_{k,j} \cdot \mathbf{u}_{i-k,j}.$$

Finally, to sum $\boldsymbol{B}^{(2)}$ we add the corresponding entry giving us

$$\left(\boldsymbol{K} * \mathbf{u} + \boldsymbol{B}^{(2)}\right) = \left(\sum_{k=0}^{i} \boldsymbol{K}_{k,j} \cdot \mathbf{u}_{i-k,j}\right) + \boldsymbol{B}^{(2)}_{i,j}. \tag{52}$$

Combining Equations (51) and (52), gives us Equation (50) as expected. □

### D.7.1 SOME PARTIAL DERIVATIVES ARE ALWAYS ZERO

To simplify future computations in this section we will state a simple lemma on chunks of the gradients of pieces of a layer that go to 0.

**Lemma D.51.** *Fix $i, j$. Then for any $j' \neq j$, $0 \leq \ell < N$, and $0 \leq k < N$ we have,*

$$\frac{\partial \boldsymbol{Z}_{i,j'}}{\partial \boldsymbol{W}_{\ell,j}} = \frac{\partial \boldsymbol{Z}_{i,j'}}{\partial \boldsymbol{K}_{k,j}} = 0.$$

*Further, any $(i', j') \neq (i, j)$ we have,*

$$\frac{\partial \boldsymbol{Z}_{i',j'}}{\partial \boldsymbol{B}^{(1)}_{i,j}} = \frac{\partial \boldsymbol{Z}_{i',j'}}{\partial \boldsymbol{B}^{(2)}_{i,j}} = 0.$$

*Proof.* Follows from Equation (50) and definition of partial derivatives. □

### D.7.2 GENERIC FORM OF PARTIAL DERIVATIVES PLUS A CONSEQUENCE

Given Lemma D.50 we can conclude the following.

**Lemma D.52.** *For $0 \leq i < N$ and $0 \leq j < d$, any entry $x \in \{\boldsymbol{W}_{i,j}, \boldsymbol{B}^{(1)}_{i,j}\}$,*

$$\frac{\partial \boldsymbol{Z}_{i,j}}{\partial x} = \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right) \frac{\partial}{\partial x} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right)$$

*and $0 \leq i < N$ and $0 \leq j < d$, any entry $x \in \{\boldsymbol{K}_{i,j}, \boldsymbol{B}^{(2)}_{i,j}\}$,*

$$\frac{\partial \boldsymbol{Z}_{i,j}}{\partial x} = \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right) \cdot \frac{\partial}{\partial x} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right).$$

A consequence of Lemma D.52 is the following.

**Corollary D.53.** *Let $\boldsymbol{\theta} = \left( \boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(1)}, \boldsymbol{B}^{(2)} \right) = \mathbf{0}$. Then for all parameter variable's x, we have*

$$\frac{\partial \boldsymbol{Z}_{i,j}}{\partial x} = 0.$$

*Specifically,*

$$\nabla_{\boldsymbol{\theta}} \overline{L}(\boldsymbol{\theta}) \,|_{\boldsymbol{\theta}=\mathbf{0}} = \mathbf{0}.$$

Corollary D.53 implies that initializing $\boldsymbol{\theta} = \mathbf{0}$ is not a good choice for initializing parameters since it is a local minima.

We can exactly figure out the partial derivatives of Lemma D.52 by the following.

**Lemma D.54.** *Fix $i, j$. Then for any $0 \leq \ell < N$ we have,*

$$\frac{\partial}{\partial \boldsymbol{W}_{\ell,j}} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right) = \mathbf{u}_{i,\ell}$$

*and*

$$\frac{\partial}{\partial \boldsymbol{B}^{(1)}_{i,j}} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right) = 1.$$

*Also,*

$$\frac{\partial}{\partial \boldsymbol{B}^{(2)}_{i,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right) = 1.$$

*Next, for any $0 \leq k \leq i$, we have*

$$\frac{\partial}{\partial \boldsymbol{K}_{k,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right) = \mathbf{u}_{i-k,j}$$

*and for all $k > i$,*

$$\frac{\partial}{\partial \boldsymbol{K}_{k,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right) = 0.$$

*We don't need to worry about $k > i$ since those pieces will also be $0$ due to the definition of convolution.*

*Proof.* Let's begin by looking at

$$\frac{\partial}{\partial \boldsymbol{W}_{\ell,j}} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right).$$

Expanding this out gives us

$$\frac{\partial}{\partial \boldsymbol{W}_{\ell,j}} \left( \sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j} + \boldsymbol{B}^{(1)}_{i,j} \right).$$

When we take the partial derivative of this with respect to $\boldsymbol{W}_{\ell,j}$, $\boldsymbol{B}^{(1)}_{i,j}$ goes to 0. And the term

$$\frac{\partial}{\partial \boldsymbol{W}_{\ell,j}} \left( \sum_{\ell=0}^{d-1} \mathbf{u}_{i,\ell} \cdot \boldsymbol{W}_{\ell,j} \right) = \mathbf{u}_{i,\ell},$$

as desired.

Next, let us look at

$$\frac{\partial}{\partial \boldsymbol{B}^{(1)}_{i,j}} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right)$$

Since $\boldsymbol{B}^{(1)}_{i,j}$ doesn't show up in the dot product of the vectors, we know that piece goes to zero, giving us

$$\frac{\partial}{\partial \boldsymbol{B}^{(1)}_{i,j}} \left( \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle + \boldsymbol{B}^{(1)}_{i,j} \right) = \frac{\partial \boldsymbol{B}^{(1)}_{i,j}}{\partial \boldsymbol{B}^{(1)}_{i,j}} = 1,$$

as desired.

Next, for any $0 \leq k \leq i$ we have

$$\frac{\partial}{\partial \boldsymbol{K}_{k,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right).$$

The $\boldsymbol{B}^{(2)}_{i,j}$ term goes to 0 as we're taking the partial derivative with respect to $\boldsymbol{K}_{k,j}$. So we have

$$\frac{\partial}{\partial \boldsymbol{K}_{k,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] \right) = \frac{\partial}{\partial \boldsymbol{K}_{k,j}} \sum_{k'=0}^{i} \boldsymbol{K}_{k',j} \mathbf{u}_{i-k',j} = \mathbf{u}_{i-k,j}$$

as desired, since $\boldsymbol{K}_{k,j}$ only shows up in the summation when $k' = k$. Then for $k > i$ we have

$$\frac{\partial}{\partial \boldsymbol{K}_{k,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] \right) = \frac{\partial}{\partial \boldsymbol{K}_{k,j}} \sum_{k'=0}^{i} \boldsymbol{K}_{k',j} \mathbf{u}_{i-k',j}, = 0 \tag{53}$$

as desired, since $\boldsymbol{K}_{k,j}$ will never show up in the summation as $k' < k$.

Finally, lets look at the fourth piece,

$$\frac{\partial}{\partial \boldsymbol{B}^{(2)}_{i,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right).$$

The term $\boldsymbol{B}^{(2)}_{i,j}$ doesn't appear in the result of the convolution operation, therefore that piece goes to 0, giving us

$$\frac{\partial}{\partial \boldsymbol{B}^{(2)}_{i,j}} \left( \left( \boldsymbol{K}_{:,j} * \mathbf{u}_{:,j} \right)[i] + \boldsymbol{B}^{(2)}_{i,j} \right) = \frac{\partial \boldsymbol{B}^{(2)}_{i,j}}{\partial \boldsymbol{B}^{(2)}_{i,j}} = 1,$$

as desired.

□

**Another assumption** Before our main result we have another restriction on the target $f$ which is that $f$ must be defined with a linear map, $\overline{\boldsymbol{W}} \in \mathbb{R}^{d \times d}$ that has non-zero columns. We make further assumptions on $\boldsymbol{W}, \boldsymbol{B}^{(1)}, \boldsymbol{K}, \boldsymbol{B}^{(2)}$ which we will justify later. We note that this assumption is satisfied for both the SQUARE and LINEAR functions.

**Assumption D.55.** For all $j$ either $(\boldsymbol{K}_{:,j} \neq \mathbf{0})$ or $\left(\boldsymbol{B}^{(2)}_{:,j} \neq \mathbf{0}\right)$ and $\boldsymbol{W}_{:,j} \neq \mathbf{0}$. Further, we have $\boldsymbol{B}^{(1)} = \mathbf{0}$.

The target function $f : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d}$ is

1. 1-layer $BC\left(\overline{\boldsymbol{W}}, \overline{\boldsymbol{K}}, \overline{\boldsymbol{B}}^{(1)}, \overline{\boldsymbol{B}}^{(2)}\right)$ such that for all $j$, $\left\|\overline{\boldsymbol{W}}_{:,j}\right\|_2 \geq 0$
2. $f(\mathbf{u})_{i,j}$ only depends on $\mathbf{u}_{i,:}$

With the above assumptions and definitions, we are finally ready to look at the expected partial derivatives of pieces of the loss function.

**Definition D.56.** For the rest of the section, we will redefine $\boldsymbol{\theta} = \left(\boldsymbol{W}, \boldsymbol{K}, \boldsymbol{B}^{(2)}\right)$. Note that we are just removing $\boldsymbol{B}^{(1)}$ since it is all zeros as per Assumption D.55.

**Lemma D.57.** *Given Assumption D.45 and recall that $\boldsymbol{B}^{(1)} = \mathbf{0}$. Fix $i, j$. Then we have*

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = \boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right] \boldsymbol{W}^2_{\ell',j}.$$

*Next, for any $0 \leq k \leq i$ we have*

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = \boldsymbol{K}_{k,j} \sum_{\ell'=0}^{d-1} \boldsymbol{W}^2_{\ell',j} \mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right] \mathbb{E}\left[\mathbf{u}^2_{i-k,j}\right].$$

*(Recall for $k > i$ partial derivatives are 0).*

*Finally, for any $0 \leq \ell \leq d-1$,*

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \boldsymbol{W}_{\ell,j} \sum_{k'=0}^{i} \boldsymbol{K}^2_{k',j} \mathbb{E}\left[\mathbf{u}^2_{i-k',j} \mathbf{u}^2_{i,\ell}\right] + \left(\boldsymbol{B}^{(2)}_{i,j}\right)^2 \boldsymbol{W}_{\ell,j} \mathbb{E}\left[\mathbf{u}^2_{i,\ell}\right].$$

*Proof.* Given Lemma D.52 and Lemma D.54 (along with the fact that $\boldsymbol{B}^{(1)} = \mathbf{0}$) we have

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{Z}_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] &= \mathbb{E}\left[\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}[i] + \boldsymbol{B}^{(2)}_{i,j}\right) \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j} \rangle^2\right] \\
&= \sum_{\ell'=0}^{d-1} \sum_{\ell''=0}^{d-1} \sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell'} \mathbf{u}_{i-k',j} \mathbf{u}_{i,\ell''}\right] \boldsymbol{W}_{\ell',j} \boldsymbol{W}_{\ell'',j} \boldsymbol{K}_{k',j} \\
&\quad + \boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1} \sum_{\ell''=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'} \mathbf{u}_{i,\ell''}\right] \boldsymbol{W}_{\ell',j} \boldsymbol{W}_{\ell'',j}.
\end{aligned}$$

In the above, the first summation goes to 0 since for all $\ell', \ell'', k$, by Assumption D.45, as the expected value of the product of three $\mathbf{u}$'s will be 0 since there's an odd number of them. Again, by Assumption D.45, the second summation will be non-zero if and only if $\ell' = \ell''$. Therefore we get the following,

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = \boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right] \boldsymbol{W}^2_{\ell',j}$$

as desired.

Moving onto the next piece, using Lemma D.52 and Lemma D.54 (along with the fact that $\boldsymbol{B}^{(1)} = \mathbf{0}$) we have

$$\begin{aligned}\mathbb{E}\left[\boldsymbol{Z}_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] &= \mathbb{E}\left[\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}[i] + \boldsymbol{B}^{(2)}_{i,j}\right)\langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j}\rangle^2 \mathbf{u}_{i-k,j}\right] \\ &= \sum_{\ell'=0}^{d-1}\sum_{\ell''=0}^{d-1}\sum_{k'=0}^{i}\mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell''}\mathbf{u}_{i-k',j}\mathbf{u}_{i-k,j}\right]\boldsymbol{W}_{\ell',j}\boldsymbol{W}_{\ell'',j}\boldsymbol{K}_{k',j} \\ &+ \boldsymbol{B}^{(2)}_{i,j}\sum_{\ell'}^{d-1}\sum_{\ell''=0}^{d-1}\mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell''}\mathbf{u}_{i-k,j}\right]\boldsymbol{W}_{\ell',j}\boldsymbol{W}_{\ell'',j}\end{aligned}$$

By Assumption D.45, only expected values of terms with square monomials are non-zero. Specifically, the first summation has the $\mathbf{u}_{i-k,j}$ term, therefore, we need $k' = k$ to get an even exponent. This is the same reasoning for $\ell' = \ell''$. Therefore, the first summation is non-zero if and only if $k' = k$ and $\ell' = \ell''$. The second summation will be 0 since for all $\ell', \ell''$ the expected value of the $\mathbf{u}$'s is 0 since there's an odd number of them, there will always be an odd exponent. So we get

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = \boldsymbol{K}_{k,j}\sum_{\ell'=0}^{d-1}\boldsymbol{W}^2_{\ell',j}\mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right]\mathbb{E}\left[\mathbf{u}^2_{i-k,j}\right]$$

as desired. In the above we note that by Assumption D.45, any two entries in $\mathbf{u}$ are independent random variables so we can multiply their expectations as above.

Moving onto the final piece, given Lemma D.52 and Lemma D.54 and $\boldsymbol{B}^{(1)} = \mathbf{0}$ we have

$$\begin{aligned}\mathbb{E}\left[\boldsymbol{Z}_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \mathbb{E}\left[\left(\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}\right)[i] + \left(\boldsymbol{B}^{(2)}_{i,j}\right)\right)^2\langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j}\rangle \mathbf{u}_{i,\ell}\right] \\ &= \sum_{k'=0}^{i}\sum_{k''=0}^{i}\sum_{\ell'=0}^{d-1}\mathbb{E}\left[\mathbf{u}_{i-k',j}\mathbf{u}_{i,\ell'}\mathbf{u}_{i-k'',j}\mathbf{u}_{i,\ell}\right]\boldsymbol{K}_{k',j}\boldsymbol{K}_{k'',j}\boldsymbol{W}_{\ell',j} \\ &+ 2\boldsymbol{B}^{(2)}_{i,j}\sum_{k'=0}^{i}\sum_{\ell'=0}^{d-1}\mathbb{E}\left[\mathbf{u}_{i-k',j}\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell}\right]\boldsymbol{K}_{k',j}\boldsymbol{W}_{\ell',j} \\ &+ \left(\boldsymbol{B}^{(2)}_{i,j}\right)^2\sum_{\ell'=0}^{d-1}\mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell}\right]\boldsymbol{W}_{\ell',j}.\end{aligned}$$

We again use Assumption D.45 to simplify teh summations. The first summation has the $\mathbf{u}_{i,\ell}$ term, therefore to get an even exponent on it we need $\ell' = \ell$. This is the same reasoning for $k' = k''$. Therefore the first summation will be non-zero if and only if $\ell' = \ell$ and $k' = k''$. The second summation will be 0 for all $k', \ell'$ since we're taking the expected value of an odd number of $\mathbf{u}$ products, there will always be an odd exponent. The third term will be non-zero if and only if $\ell' = \ell$ to get an even exponent on $\mathbf{u}$'s entry. Therefore we have,

$$\mathbb{E}\left[\boldsymbol{Z}_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \boldsymbol{W}_{\ell,j}\sum_{k'=0}^{i}\boldsymbol{K}^2_{k',j}\mathbb{E}\left[\mathbf{u}^2_{i-k',j}\right]\mathbb{E}\left[\mathbf{u}^2_{i,\ell}\right] + \left(\boldsymbol{B}^{(2)}_{i,j}\right)^2\boldsymbol{W}_{\ell,j}\mathbb{E}\left[\mathbf{u}^2_{i,\ell}\right]$$

as desired. In the above we note that by Assumption D.45, any two entries in $\mathbf{u}$ are independent random variables.

□

**Lemma D.58.** *Fix $i, j$. Then we have*

$$\mathbb{E}\left[\mathbf{u}^2_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = 0.$$

*Next, we have*

$$\mathbb{E}\left[\mathbf{u}^2_{i,j}\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{0,j}}\right] = \boldsymbol{W}_{j,j}\mathbb{E}\left[\mathbf{u}^4_{i,j}\right].$$

*For $k > 0$, we have*

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = 0.$$

*Next,*

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{j,j}}\right] = \boldsymbol{K}_{0,j}\mathbb{E}\left[\mathbf{u}_{i,j}^4\right].$$

*otherwise, when $\ell \neq j$ we have*

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = 0.$$

*Proof.* Given Lemma D.52 and Lemma D.54 we have

$$\begin{aligned}\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] &= \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j}\rangle \mathbf{u}_{i,j}^2 \\ &= \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,j}^2\right] \boldsymbol{W}_{\ell',j}.\end{aligned}$$

We simplify the above using Assumption D.45. Therefore, the summation is 0 for all $\ell$ since we're taking the expected value of an odd number of $\mathbf{u}$'s. Therefore,

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = 0.$$

Next, by Lemma D.52 and Lemma D.54 we have

$$\begin{aligned}\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] &= \langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,j}\rangle \mathbf{u}_{i-k,j}\mathbf{u}_{i,j}^2 \\ &= \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,j}^2\mathbf{u}_{i-k,j}\mathbf{u}_{i,\ell'}\right] \boldsymbol{W}_{\ell',j}.\end{aligned}$$

We simplify the above using Assumption D.45. The summation will be non-zero only when $k = 0$ and $\ell' = j$ to ensure we have an even exponent on the $\mathbf{u}$ variable, since that is the only way to get even exponents on the $\mathbf{u}$'s. Therefore we have

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{0,j}}\right] = \boldsymbol{W}_{j,j}\mathbb{E}\left[\mathbf{u}_{i,j}^4\right],$$

and for $k > 0$,

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = 0.$$

Moving onto the final piece, given Lemma D.52 and Lemma D.54, we have

$$\begin{aligned}\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \left(\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}[i] + \boldsymbol{B}_{i,j}^{(2)}\right)\right) \mathbf{u}_{i,\ell}\mathbf{u}_{i,j}^2 \\ &= \sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i-k',j}\mathbf{u}_{i,\ell}\mathbf{u}_{i,j}^2\right] \boldsymbol{K}_{k',j} + \boldsymbol{B}_{i,j}^{(2)}\mathbb{E}\left[\mathbf{u}_{i,\ell}\mathbf{u}_{i,j}^2\right].\end{aligned}$$

We simplify the above using Assumption D.45. The summation will be non-zero only when $k' = 0$ and $\ell = j$, since that is the only way to get even exponents on the $\mathbf{u}$'s. The second term will be 0 since we're taking the expected value of a non-square monomial. Therefore we have,

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{j,j}}\right] = \boldsymbol{K}_{0,j}\mathbb{E}\left[\mathbf{u}_{i,j}^4\right].$$

Otherwise, when $\ell \neq j$ we have,

$$\mathbb{E}\left[\mathbf{u}_{i,j}^2 \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = 0$$

as desired. □

**Theorem D.59.** *Given Assumptions D.45, D.46, D.55, and a function*

$$f(\mathbf{u}) = \mathbf{u} \odot \mathbf{u}.$$

*Let $\boldsymbol{\theta}$ be such that, $\mathbb{E}\nabla_{\boldsymbol{\theta}}\overline{L} = \mathbf{0}$ then* $\textsc{BaseConv}(\mathbf{u}, \boldsymbol{\theta}) = f(\mathbf{u})$.

*Proof.* Recall our loss function from Equation (49). Then given Lemma D.57 and Lemma D.58 we know that for any $k \geq 0$,

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{k,j}}\right] &= \sum_{i'j'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j'}}{\partial \boldsymbol{K}_{k,j}}\right] \\ &= \sum_{i'} \mathbb{E}\left[\left(\boldsymbol{Z}_{i',j} - \mathbf{u}^2_{i',j}\right)\frac{\partial \boldsymbol{Z}_{i',j}}{\partial \boldsymbol{K}_{k,j}}\right].\end{aligned}$$

In the above, the second equality follows from Lemma D.51.

From Lemma D.57, Lemma D.58 we have the following for $k > 0$,

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{k,j}}\right] = \boldsymbol{K}_{k,j} \sum_{i'} \sum_{\ell'=0}^{d-1} \boldsymbol{W}^2_{\ell',j} \mathbb{E}\left[\mathbf{u}^2_{i',\ell'}\right] \mathbb{E}\left[\mathbf{u}^2_{i'-k,j}\right].$$

We know that from Assumption D.46 that the expected value of $\mathbb{E}\left[\mathbf{u}^2_{i',\ell'}\right]$ and $\mathbb{E}\left[\mathbf{u}^2_{i'-k,j}\right]$ are strictly positive due to having an even exponent. Also, due to Assumption D.55 we know that $\boldsymbol{W}_{:,j}$ has at least one non-zero entry for all $j$. This implies that for at least one $\ell'$, $\boldsymbol{W}^2_{\ell',j} > 0$ (and hence the summation is strictly a positive value).Therefore to set this partial derivative to 0 implies that $\boldsymbol{K}_{k,j} = 0$. Explicitly, $\boldsymbol{K}_{k,j} = 0$ for all $j$ and $k > 0$. This in turn implies that for all $j$:

$$\boldsymbol{K}_{:,j} \neq \mathbf{0} \Leftrightarrow \boldsymbol{K}_{0,j} \neq 0. \tag{54}$$

Next we'll examine the loss function when we take the partial derivative with respect to $\boldsymbol{W}_{\ell,j}$. For all $\ell \neq j$,

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \sum_{i'j'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j'}}{\partial \boldsymbol{W}_{\ell,j}}\right] \\ &= \sum_{i'} \left(\boldsymbol{W}_{\ell,j} \sum_{k'=0}^{i'} \boldsymbol{K}^2_{k',j} \mathbb{E}\left[\mathbf{u}^2_{i'-k',j}\mathbf{u}^2_{i',\ell}\right] + \left(\boldsymbol{B}^{(2)}_{i',j}\right)^2 \boldsymbol{W}_{\ell,j} \mathbb{E}\left[\mathbf{u}^2_{i',\ell}\right]\right).\end{aligned}$$

In the above the second equality follows from Lemma D.51, Lemma D.57, and Lemma D.58.

We know from above that for $k > 0$, $\boldsymbol{K}_{k,j} = 0$. Thus, we can simplify the above to

$$\begin{aligned}&\sum_{i'} \boldsymbol{W}_{\ell,j} \left(\boldsymbol{K}^2_{0,j} \mathbb{E}\left[\mathbf{u}^2_{i',j}\right] \mathbb{E}\left[\mathbf{u}^2_{i',\ell}\right] + \left(\boldsymbol{B}^{(2)}_{i',j}\right)^2 \mathbb{E}\left[\mathbf{u}^2_{i',\ell}\right]\right) \\ &= \boldsymbol{W}_{\ell,j} \sum_{i'} \mathbb{E}\left[\mathbf{u}^2_{i',\ell}\right] \left(\boldsymbol{K}^2_{0,j} \mathbb{E}\left[\mathbf{u}^2_{i',j}\right] + \left(\boldsymbol{B}^{(2)}_{i',j}\right)^2\right).\end{aligned}$$

By Equation (54) we know $\boldsymbol{K}_{0,j} > 0$ and from Assumption D.55 at least one entry in each column of $\boldsymbol{B}^{(2)}$ may be non-zero, $\boldsymbol{B}^{(2)}_{:,j} > 0$. Therefore

$$\left(\boldsymbol{K}^2_{0,j} \mathbb{E}\left[\mathbf{u}^2_{i',j}\right] + \boldsymbol{B}^{(2)}_{i',j}\right) \neq 0 \implies \boldsymbol{W}_{\ell,j} = 0$$

Since we proved that $\boldsymbol{W}_{\ell,j} = 0$ for all $\ell \neq j$, and we know at least one entry in each column $\boldsymbol{W}_{:,j} \neq \mathbf{0}$, we must have

$$\boldsymbol{W}_{j,j} \neq 0. \tag{55}$$

Moving onto $\boldsymbol{B}^{(2)}_{i,j}$, we have

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{B}^{(2)}_{i,j}}\right] = \boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1} \boldsymbol{W}^2_{\ell',j} \mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right].$$

The above summation piece is not equal to zero as we know $\boldsymbol{W}_{j,j} \neq 0$ and hence, $\boldsymbol{W}^2_{j,j} > 0$. Also note the exponents on the $\mathbf{u}$'s is even, and so by Assumption D.45 we have for all $\ell'$, $\mathbb{E}\left[\mathbf{u}^2_{i,\ell'}\right] > 0$. Therefore, if we set

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{B}^{(2)}_{i,j}}\right] = 0$$

we get for all $i, j$,

$$\boldsymbol{B}^{(2)}_{i,j} = 0.$$

Now that we know $\boldsymbol{B}^{(2)} = \mathbf{0}$ this will help us simplify the following. Given Lemma D.57 and Lemma D.58, for $\ell = j$, we have

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{j,j}}\right] = \sum_{i'} \boldsymbol{W}_{j,j} \boldsymbol{K}^2_{0,j} \mathbb{E}\left[\mathbf{u}^4_{i',j}\right] - \boldsymbol{K}_{0,j} \mathbb{E}\left[\mathbf{u}^4_{i',j}\right],$$

we simplify and remove $\boldsymbol{B}^{(2)}$ terms and get

$$\boldsymbol{K}_{0,j}\left(\boldsymbol{K}_{0,j}\boldsymbol{W}_{j,j} - 1\right)\left(\sum_{i'} \mathbb{E}\left[\mathbf{u}^4_{i',j}\right]\right).$$

Setting this to 0 we can conclude that since $\boldsymbol{K}_{0,j} \neq 0$ and the above summation isn't 0, we get $\boldsymbol{K}_{0,j}\boldsymbol{W}_{j,j} = 1$.

So far we have that $\boldsymbol{B}^{(2)} = \mathbf{0}$ and $\boldsymbol{K}_{k,j} = 0$ for $k > 0$. Also that $\boldsymbol{K}_{0,j} \neq 0$ for all $j$. Then by Lemma D.58 and Lemma D.57, consider

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{0,j}}\right] &= \sum_{i'} \left(\boldsymbol{K}_{0,j}\boldsymbol{W}^2_{j,j}\mathbb{E}\left[\mathbf{u}^2_{i',j}\mathbf{u}^2_{i',j}\right] - \boldsymbol{W}_{j,j}\left[\mathbf{u}^4_{i',j}\right]\right) \\ &= \boldsymbol{W}_{j,j}\left(\boldsymbol{K}_{0,j}\boldsymbol{W}_{j,j} - 1\right)\left(\sum_{i'} \mathbb{E}\left[\mathbf{u}^4_{i',j}\right]\right). \end{aligned} \tag{56}$$

We know from (55) that $\boldsymbol{W}_{j,j} \neq 0$ and the summation piece isn't 0 by Assumption D.45. Therefore,

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{0,j}}\right] = 0 &\implies \boldsymbol{K}_{0,j} \cdot \boldsymbol{W}_{j,j} - 1 = 0 \\ &\implies \boldsymbol{K}_{0,j} \cdot \boldsymbol{W}_{j,j} = 1\end{aligned}$$

as desired.

Note that this implies for all $j$:

$$\boldsymbol{K}_{0,j} = \frac{1}{\boldsymbol{W}_{j,j}}.$$

Given Assumption D.55 and the above values; we get that $\boldsymbol{W}$ is a diagonal matrix, $\boldsymbol{K} = \begin{pmatrix} \frac{1}{\boldsymbol{W}_{0,0}} & \frac{1}{\boldsymbol{W}_{1,1}} \cdots \frac{1}{\boldsymbol{W}_{d-1,d-1}} \\ \mathbf{0}^{N-1\times d-1} \end{pmatrix}$, $\boldsymbol{B}^{(2)} = \mathbf{0}^{N\times d}$. Lets use these pieces to show that $\textsc{BaseConv}(\mathbf{u}) = \mathbf{u} \odot \mathbf{u}$ (recall that $\boldsymbol{B}^{(1)} = \mathbf{0}$).

Indeed,

$$\textsc{BaseConv}(\mathbf{u}) = (\mathbf{u} \cdot \boldsymbol{W}) \odot \left(\boldsymbol{K} * \mathbf{u} + \boldsymbol{B}^{(2)}\right)$$

Plugging in our values we get

$$\left(\mathbf{u}\cdot\begin{pmatrix}\boldsymbol{W}_{0,0} & 0 & \dots & 0\\ 0 & \boldsymbol{W}_{1,1} & \dots & 0\\ & & & \\ 0 & 0 & \ddots & 0\\ 0 & 0 & \dots & \boldsymbol{W}_{d-1,d-1}\end{pmatrix}\right)\odot\left(\begin{pmatrix}\frac{1}{\boldsymbol{W}_{0,0}} & \frac{1}{\boldsymbol{W}_{1,1}}\cdots\frac{1}{\boldsymbol{W}_{d-1,d-1}}\\ & \mathbf{0}^{N-1\times d-1}\end{pmatrix}*\mathbf{u}+\mathbf{0}^{N\times d}\right).$$

Note that

$$\boldsymbol{K}*\mathbf{u}=\mathbf{u}\begin{pmatrix}\frac{1}{\boldsymbol{W}_{0,0}} & 0 & \dots & 0\\ 0 & \frac{1}{\boldsymbol{W}_{1,1}} & \dots & 0\\ & & & \\ 0 & 0 & \ddots & 0\\ 0 & 0 & \dots & \frac{1}{\boldsymbol{W}_{d-1,d-1}}\end{pmatrix}=\mathbf{u}\boldsymbol{W}^{-1},$$

which gives us

$$(\mathbf{u}\boldsymbol{W}\odot\boldsymbol{K}*\mathbf{u})=\left(\mathbf{u}\boldsymbol{W}\odot\mathbf{u}\boldsymbol{W}^{-1}\right)=\mathbf{u}\odot\mathbf{u},$$

where the last equality follows since $\boldsymbol{W}$ is a diagonal matrix with all non-zero entries in it's diagonal. □

The following lemma will be for when the function we are considering is an linear map.

**Lemma D.60.** *Fix $\overline{\boldsymbol{W}}\in\mathbb{R}^{d\times d}$ and $i,j$. Then we have*

$$\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{B}^{(2)}{}_{i,j}}\right]=\sum_{\ell}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell,j}\mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right].$$

*For all $k$,*

$$\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{K}_{k,j}}\right]=0.$$

*For all $\ell$,*

$$\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{W}_{\ell,j}}\right]=\boldsymbol{B}^{(2)}_{i,j}\overline{\boldsymbol{W}}_{\ell,j}\mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right].$$

*Proof.* Given Lemma D.52 and Lemma D.54 (and the fact that $\boldsymbol{B}^{(1)}=\mathbf{0}$) we have

$$\begin{aligned}\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{B}^{(2)}{}_{i,j}}\right]&=\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\boldsymbol{W}_{:,j}\right\rangle\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\right]\\&=\sum_{\ell=0}^{d-1}\sum_{\ell'=0}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell',j}\mathbb{E}\left[\mathbf{u}_{i,\ell}\mathbf{u}_{i,\ell'}\right]\end{aligned}$$

From Assumption D.45 we get that the summation will always be non-zero if and only if $\ell=\ell'$ so that the $\mathbf{u}$ variable has an even exponent. Therefore we get,

$$\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{B}^{(2)}{}_{i,j}}\right]=\sum_{\ell=0}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell,j}\mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right]$$

as desired.

Next, for all $k$, by Lemma D.52 and Lemma D.54 (and the fact that $\boldsymbol{B}^{(1)}=\mathbf{0}$)

$$\begin{aligned}\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\frac{\partial\boldsymbol{Z}_{i,j}}{\partial\boldsymbol{K}_{k,j}}\right]&=\mathbb{E}\left[\left\langle\mathbf{u}_{i,:}\boldsymbol{W}_{:,j}\right\rangle\mathbf{u}_{i-k,j}\left\langle\mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle\right]\\&=\sum_{\ell=0}^{d-1}\sum_{\ell'=0}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell',j}\mathbb{E}\left[\mathbf{u}_{i,\ell}\mathbf{u}_{i,\ell'}\mathbf{u}_{i-k,j}\right].\end{aligned}$$

We simplify the above using Assumption D.45. The summation goes to 0 since there are an odd number of $\mathbf{u}$ terms, there will always be an odd exponent. Note that this is true for $k \leq i$. Recall from Equation (53) that for $k > i$,

$$\frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}} = 0.$$

Therefore, for all $k$,

$$\mathbb{E}\left[\left\langle \mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = 0$$

as desired.

Next, for all $\ell$, by Lemma D.52 and Lemma D.54 (and the fact that $\boldsymbol{B}^{(1)} = \mathbf{0}$)

$$\begin{aligned}\mathbb{E}\left[\left\langle \mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \mathbb{E}\left[\left(\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}\right)[i] + \boldsymbol{B}^{(2)}_{i,j}\right)\mathbf{u}_{i,\ell}\left\langle \mathbf{u}_{i,:}, \overline{\boldsymbol{W}}_{:,j}\right\rangle\right] \\ &= \left(\sum_{k'=0}^{i}\sum_{\ell'=0}^{d-1} \boldsymbol{K}_{k,j}\overline{\boldsymbol{W}}_{\ell',j}\mathbb{E}\left[\mathbf{u}_{i-k,j}\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell}\right]\right) \\ &+ \boldsymbol{B}^{(2)}_{i,j}\sum_{\ell'=0}^{d-1}\overline{\boldsymbol{W}}_{\ell',j}\mathbb{E}\left[\mathbf{u}_{i,\ell}\mathbf{u}_{i,\ell'}\right]\end{aligned}$$

We simplify the above using Assumption D.45. The first summation will always be 0 due to an odd exponent on the $\mathbf{u}$'s. The second summation piece will always be non-zero if and only if $\ell' = \ell$, giving us the even exponent on the $\mathbf{u}$ variable. Therefore we get,

$$\mathbb{E}\left[\left\langle \mathbf{u}_{i,:}\overline{\boldsymbol{W}}_{:,j}\right\rangle \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \boldsymbol{B}^{(2)}_{i,j}\overline{\boldsymbol{W}}_{\ell,j}\mathbb{E}\left[\mathbf{u}^2_{i,\ell}\right]$$

as desired. □

We make another assumption to assist with the following theorem on linear maps.

**Assumption D.61.** For all $j$, $\left\langle \boldsymbol{W}_{:,j}, \overline{\boldsymbol{W}}_{:,j}\right\rangle \neq 0$ and $\overline{\boldsymbol{W}}_{:,j} \neq \mathbf{0}$.

**Theorem D.62.** *Given Assumptions D.45, D.46, D.55, D.61, and a function*

$$f(\mathbf{u}) = \mathbf{u}\overline{\boldsymbol{W}}.$$

*Let $\boldsymbol{\theta}$ be such that, $\mathbb{E}\nabla_{\boldsymbol{\theta}}\overline{L} = \mathbf{0}$ then* BASECONV$(\mathbf{u}, \boldsymbol{\theta}) = f(\mathbf{u})$.

*Proof.* From lemma D.51 we get that

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{B}^{(2)}_{i,j}}\right] &= \sum_{i',j'}\mathbb{E}\left[\frac{\partial \overline{L}_{i',j'}}{\partial \boldsymbol{B}^{(2)}_{i',j'}}\right] \\ &= \sum_{i,j}\mathbb{E}\left[\frac{\partial \overline{L}_{i,j}}{\partial \boldsymbol{B}^{(2)}_{i,j}}\right]\end{aligned}$$

Recall our loss function from Equation (49). Then given Lemma D.57 and Lemma D.60 we know that

$$\sum_{i,j}\mathbb{E}\left[\frac{\partial \overline{L}_{i,j}}{\partial \boldsymbol{B}^{(2)}_{i,j}}\right] = \boldsymbol{B}^{(2)}_{i,j}\sum_{\ell=0}^{d-1}\boldsymbol{W}^2_{\ell,j}\mathbb{E}\left[\mathbf{u}^2_{i,\ell}\right] - \sum_{\ell=0}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell,j}\mathbb{E}\left[\mathbf{u}^2_{i,j}\right] = 0,$$

Which implies that

$$\boldsymbol{B}^{(2)}_{i,j} = \frac{\sum_{\ell=0}^{d-1}\boldsymbol{W}_{\ell,j}\overline{\boldsymbol{W}}_{\ell,j}}{\sum_{\ell=0}^{d-1}\boldsymbol{W}^2_{\ell,j}}.$$

Given Assumption D.61 we know that the numerator will be non-zero and given assumption Assumption D.55 we know the denominator will always be non-zero as well. Therefore we get that for all $i, j$

$$\boldsymbol{B}_{i,j} = \frac{\langle \boldsymbol{W}_{:,j}, \overline{\boldsymbol{W}}_{:,j} \rangle}{\langle \boldsymbol{W}_{:,j}, \boldsymbol{W}_{:,j} \rangle} \stackrel{\text{def}}{=} b_j. \tag{57}$$

Next, via Lemma D.51 we have for all $k \geq 0$:

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{k,j}}\right] &= \sum_{i',j'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j'}}{\partial \boldsymbol{K}_{k,j'}}\right] \\ &= \sum_{i'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j}}{\partial \boldsymbol{K}_{k,j}}\right]\end{aligned}$$

Then from Lemma D.57 and lemma D.60 we get

$$\sum_{i'} \mathbb{E}\left[\boldsymbol{Z}_{i',j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = \sum_{i'} \boldsymbol{K}_{k,j} \sum_{\ell'=0}^{d-1} \boldsymbol{W}_{\ell',j}^2 \mathbb{E}\left[\mathbf{u}_{i',\ell'}^2\right] \mathbb{E}\left[\mathbf{u}_{i'-k,j}^2\right].$$

Since we know that the summation piece over $\ell'$ is always non-zero due to the even exponents on the $\mathbf{u}$ terms and at least one of $\boldsymbol{W}_{\ell',j} \neq 0$. Therefore, if we set

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{k,j}}\right] = 0,$$

we get for all $k$, $\boldsymbol{K}_{k,j} = 0$. In other words,

$$\boldsymbol{K} = \mathbf{0}^{N \times d}.$$

Finally, via Lemma D.51 we have for all $\ell$:

$$\begin{aligned}\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \sum_{i'j'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j'}}{\partial \boldsymbol{W}_{\ell,j}}\right] \\ &= \sum_{i'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j}}{\partial \boldsymbol{W}_{\ell,j}}\right]\end{aligned}$$

From Lemma D.57 and Lemma D.60 we get

$$\begin{aligned}\sum_{i'} \mathbb{E}\left[\frac{\partial \overline{L}_{i',j}}{\partial \boldsymbol{W}_{\ell,j}}\right] &= \sum_{i'} \boldsymbol{W}_{\ell,j} \sum_{k'=0}^{i'} \boldsymbol{K}_{k',j}^2 \mathbb{E}\left[\mathbf{u}_{i'-k',j}^2 \mathbf{u}_{i',\ell}^2\right] + \left(\boldsymbol{B}_{i',j}^{(2)}\right)^2 \boldsymbol{W}_{\ell,j} \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right] - \boldsymbol{B}_{i',j}^2 \overline{\boldsymbol{W}}_{\ell,j} \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right] \\ &= \boldsymbol{W}_{\ell,j} \sum_{i'} \left(\boldsymbol{B}_{i',j}^{(2)}\right)^2 \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right] - \overline{\boldsymbol{W}}_{\ell,j} \sum_{i'} \boldsymbol{B}_{i',j}^{(2)} \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right].\end{aligned}$$

We can drop the summation with $\boldsymbol{K}$ in it as we know $\boldsymbol{K} = \mathbf{0}$ Recall that from (57), $\boldsymbol{B}_{i,j}^{(2)} = b_j$. Then we can rewrite the above as

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{\ell,j}}\right] = b_j \left(\sum_{i'} \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right]\right) \left(\boldsymbol{W}_{\ell,j} b_j - \overline{\boldsymbol{W}}_{\ell,j}\right)$$

we know from Assumption D.45 that the first summation will always be non-zero since there's an even exponent on the $\mathbf{u}$ variable and we know that $b_j$ is non-zero. Therefore setting

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{\ell,j}}\right] = 0$$

tells us that

$$\boldsymbol{W}_{\ell,j} = \frac{\overline{\boldsymbol{W}}_{\ell,j}}{b_j}.$$

Given the above value for $\boldsymbol{W}_{\ell,j}$ and recall we have $\boldsymbol{K} = \mathbf{0}$ and $\boldsymbol{B}^{(2)} = (\mathbf{b_0 b_1 \dots b_{d-1}})$ where each $\mathbf{b}_j$ is a column vector comprised of all $b_j$ values. Therefore, when we take $\textsc{BaseConv}(\mathbf{u})$ we get

$$\begin{aligned}\textsc{BaseConv}(\mathbf{u}) &= (\mathbf{u}\boldsymbol{W}) \odot \left(\mathbf{0}^{N\times d} * \mathbf{u} + \boldsymbol{B}^{(2)}\right) \\ &= (\mathbf{u}\boldsymbol{W}) \odot \left(\boldsymbol{B}^{(2)}\right)\end{aligned}$$

We can rewrite $\boldsymbol{B}^{(2)}$ as

$$\boldsymbol{B}^{(2)} = \left(\mathbf{1}^{N\times d}\right) \begin{pmatrix} b_0 & 0 & \dots & 0 \\ 0 & b_1 & \dots & 0 \\ & & \ddots & \\ 0 & 0 & \dots & b_{d-1} \end{pmatrix}.$$

Lets call this diagonal matrix on the right, $\boldsymbol{D}$. Then note that

$$\boldsymbol{W} = \overline{\boldsymbol{W}}\boldsymbol{D}^{-1}.$$

Therefore, we have

$$\begin{aligned}\textsc{BaseConv}(\mathbf{u}) &= \mathbf{u}\overline{\boldsymbol{W}}\boldsymbol{D}^{-1} \odot \mathbf{1}^{N\times d}\boldsymbol{D} \\ &= \mathbf{u}\overline{\boldsymbol{W}} \odot \mathbf{1}^{N\times d} \\ &= \mathbf{u}\overline{\boldsymbol{W}},\end{aligned}$$

as desired. In the above the second inequality follows since $\boldsymbol{D}$ is a diagonal matrix. □

We now revisit the importance of Assumption D.55. Specifically, the following definition is a stronger version of the complement of Assumption D.55. The following essentially states that there are many ways to get the expected gradients of the loss function to be 0, though this doesn't imply that we have learned the exact solution, as we recover in theorem D.59.

**Definition D.63.** Define $(\text{assumption})^{\complement}$ to be

- $\boldsymbol{B}^{(2)} = \mathbf{0}$
- For all $j$, either
  - (i) $\boldsymbol{W}_{:,j} = \mathbf{0}$ and $\boldsymbol{K}_{0,j} = 0$
  - (ii) $\boldsymbol{K}_{:,j} = \mathbf{0}$ and $\boldsymbol{W}_{j,j} = 0$

The following theorem is to emphasize, there are many ways to get expected value of the gradients of the loss function to be 0.

**Theorem D.64.** *Let $\theta^*$ satisfy Definition D.63 Then $\mathbb{E}\nabla_\theta \overline{L}\big|_{\theta\leftarrow\theta^*} = 0$ when $f(\mathbf{u}) = \mathbf{u} \odot \mathbf{u}$*

*Proof.* Next, via Lemma D.57 and Lemma D.58 when $k > 0$ we have

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{k,j}}\right] = \boldsymbol{K}_{k,j} \sum_{i'} \sum_{\ell'=0}^{d-1} \boldsymbol{W}_{\ell',j}^2 \mathbb{E}\left[\mathbf{u}_{i',\ell'}^2\right] \mathbb{E}\left[\mathbf{u}_{i'-k,j}^2\right].$$

This expected value goes to zero since every column $j$ either $\boldsymbol{K}_{:,j}$ or $\boldsymbol{W}_{:,j}$ is $\mathbf{0}$.

When $k = 0$ we have

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{K}_{0,j}}\right] = \boldsymbol{W}_{j,j}\left(\boldsymbol{K}_{0,j}\boldsymbol{W}_{j,j} - 1\right)\left(\sum_{i'} \mathbb{E}\left[\mathbf{u}_{i',j}^4\right]\right).$$

This expected value goes to zero since in both D.63 and D.63 $\boldsymbol{K}_{0,j} = \boldsymbol{W}_{j,j} = 0$.

Next we have for all $\ell, j$ where $\ell \neq j$,

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \boldsymbol{W}_{\ell,j} \sum_{i'} \mathbb{E}\left[\mathbf{u}_{i',\ell}^2\right]\left(\boldsymbol{K}_{0,j}^2 \mathbb{E}\left[\mathbf{u}_{i',j}^2\right] + \boldsymbol{B}_{i',j}^{(2)}\right)$$

This expected value goes to zero since column $j$ either $\boldsymbol{K}_{:,j}$ or $\boldsymbol{W}_{:,j}$ is $\mathbf{0}$.

Then when $\ell = j$ we have

$$\mathbb{E}\left[\frac{\partial \overline{L}}{\partial \boldsymbol{W}_{j,j}}\right] = \sum_{i'} \boldsymbol{W}_{j,j}\boldsymbol{K}_{0,j}^2 \mathbb{E}\left[\mathbf{u}_{i',j}^4\right] - \boldsymbol{K}_{0,j}\mathbb{E}\left[\mathbf{u}_{i',j}^4\right]$$

This expected value goes to zero since in both (i) and (ii), $\boldsymbol{K}_{0,j} = \boldsymbol{W}_{j,j} = 0$

And we know that since $\boldsymbol{B}^{(2)}$ is all zeros, we don't need to consider the gradient of the loss function to it. □

What the above proves is that there are infinite instantiations of parameters such that the expected gradient loss is 0. However not that in *Definition D.63*, for all $j$ either $\boldsymbol{K}_{k,j}$ for $k \neq 0$ or $\boldsymbol{W}_{\ell,j}$ for $\ell \neq j$ are unconstrained. In other words, we can set these values arbitrarily, which means we get $\overline{L} \to \inf$ but we still have the expected gradient loss to be $\mathbf{0}$. This shows that some form of Assumption D.55 is necessary to prove Theorem D.59.

### D.8 Learning any possible function

We hope to be able to show that one layer of BaseConv will be able to exactly recover any one layer BaseConv, given some assumptions on the parameters. We will later explain why these assumptions are necessary for a meaningful result.

**Lemma D.65.** *Fix $i, j$. Then we have*

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = \overline{\boldsymbol{B}_{i,j}^{(2)}} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}^2\right] \overline{\boldsymbol{W}_{\ell',j}} \boldsymbol{W}_{\ell',j}.$$

*For $0 \leq k \leq i$, we have*

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = \overline{\boldsymbol{K}_{k',j}} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}^2\right] \left[\mathbf{u}_{i-k,j}^2\right] \overline{\boldsymbol{W}_{\ell',j}} \boldsymbol{W}_{\ell',j}$$

*Next for all $0 \leq \ell < d$,*

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \overline{\boldsymbol{W}_{\ell,j}} \sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right] \mathbb{E}\left[\mathbf{u}_{i-k',j}^2\right] \overline{\boldsymbol{K}_{k',j}} \boldsymbol{K}_{k',j} + \overline{\boldsymbol{B}_{i,j}^{(2)}} \boldsymbol{B}_{i,j}^{(2)} \overline{\boldsymbol{W}_{\ell,j}} \mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right]$$

*Proof.* Lets begin by looking at

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right].$$

From lemma D.52 and lemma D.54 we have

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = \mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \cdot \left(\langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,\ell} \rangle\right)\right]$$

This expands to

$$\sum_{\ell'=0}^{d-1} \sum_{\ell''=0}^{d-1} \sum_{k=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell'} \mathbf{u}_{i,\ell''} \mathbf{u}_{i-k,j}\right] \overline{\boldsymbol{W}_{\ell',j}} \boldsymbol{W}_{\ell'',j} \boldsymbol{K}_{k,j} + \overline{\boldsymbol{B}_{i,j}^{(2)}} \sum_{\ell'=0}^{d-1} \sum_{\ell''=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'} \mathbf{u}_{i,\ell''}\right] \overline{\boldsymbol{W}_{\ell',j}} \boldsymbol{W}_{\ell'',j}.$$

We simplify the above using Assumption D.45. The first summation goes to 0 as we're taking the expectation of an odd number of $\mathbf{u}$'s. The second summation will be non-zero if and only if $\ell' = \ell''$, as that gives us a squared value on the $\mathbf{u}$ variable. Therefore we get

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{B}^{(2)}{}_{i,j}}\right] = \overline{\boldsymbol{B}_{i,j}^{(2)}} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}^2\right] \overline{\boldsymbol{W}_{\ell',j}} \boldsymbol{W}_{\ell',j}$$

as desired.

Next lets consider, for all $0 \leq k \leq i$,

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right].$$

Recall we are only considering these values of $k$ as we know from Lemma D.54 that for $k > i$, the partial derivative piece is always 0. We can simplify this to the following via lemma D.52 and lemma D.54,

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \left(\langle \mathbf{u}_{i,:}, \boldsymbol{W}_{:,\ell}\rangle\right) \mathbf{u}_{i-k,j}\right]$$

This expands to

$$\sum_{\ell'=0}^{d-1}\sum_{\ell''=0}^{d-1}\sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell''}\mathbf{u}_{i-k',j}\mathbf{u}_{i-k,j}\right] \overline{\boldsymbol{W}_{\ell',j}}\boldsymbol{W}_{\ell'',j}\overline{\boldsymbol{K}_{k',j}} + \overline{\boldsymbol{B}^{(2)}_{i,j}} \sum_{\ell'=0}^{d-1}\sum_{\ell''=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell''}\mathbf{u}_{i-k,j}\right] \overline{\boldsymbol{W}_{\ell',j}}\boldsymbol{W}_{\ell'',j}$$

We simplify the above using Assumption D.45. The first summation will be non-zero if and only if the exponents on the $\mathbf{u}$'s is even. To get this we require $\ell' = \ell''$ and $k' = k$. The second summation will always be 0 as we have an odd exponent on the $\mathbf{u}$'s. Therefore we get

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{K}_{k,j}}\right] = \overline{\boldsymbol{K}_{k',j}} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}^2\right]\left[\mathbf{u}_{i-k,j}^2\right] \overline{\boldsymbol{W}_{\ell',j}}\boldsymbol{W}_{\ell',j}$$

as desired.

Finally, lets consider

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right]$$

From lemma D.52 and lemma D.54 we get

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j}\right] \left(\left(\boldsymbol{K}_{:,j} * \mathbf{u}_{:,j}\right)[i] + \boldsymbol{B}^{(2)}_{i,j}\right) \mathbf{u}_{i,\ell}$$

This expands to

$$\sum_{\ell'=0}^{d-1}\sum_{k'=0}^{i}\sum_{k''=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell}\mathbf{u}_{i-k',j}\mathbf{u}_{i-k'',j}\right] \overline{\boldsymbol{W}_{\ell',j}\boldsymbol{K}_{k',j}}\boldsymbol{K}_{k'',j} + \overline{\boldsymbol{B}^{(2)}_{i,j}} \sum_{\ell'=0}^{d-1}\sum_{k''=0}^{i} \mathbb{E}\left[\mathbf{u}_{i-k'',j}\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell'}\right] \overline{\boldsymbol{W}_{\ell',j}}\boldsymbol{K}_{k'',j}$$
$$+ \boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1}\sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i-k',j}\mathbf{u}_{i,\ell}\right] \overline{\boldsymbol{W}_{\ell',j}\boldsymbol{K}_{k',j}} + \overline{\boldsymbol{B}^{(2)}_{i,j}}\boldsymbol{B}^{(2)}_{i,j} \sum_{\ell'=0}^{d-1} \mathbb{E}\left[\mathbf{u}_{i,\ell'}\mathbf{u}_{i,\ell}\right] \overline{\boldsymbol{W}_{\ell',j}}.$$

We simplify the above using Assumption D.45. The first summation will be non-zero as we get squared exponents on the $\mathbf{u}$'s if and only if $\ell' = \ell$ and $k' = k''$. The second and third summation will always be 0 since we're taking the expectation of an odd number of $\mathbf{u}$'s. The fourth summation will be non-zero if and only if $\ell' = \ell$. Therefore we get,

$$\mathbb{E}\left[\left(\left(\mathbf{u}\overline{\boldsymbol{W}}\right) \odot \left(\overline{\boldsymbol{K}} * \mathbf{u} + \overline{\boldsymbol{B}^{(2)}}\right)\right)_{i,j} \frac{\partial \boldsymbol{Z}_{i,j}}{\partial \boldsymbol{W}_{\ell,j}}\right] = \overline{\boldsymbol{W}_{\ell,j}} \sum_{k'=0}^{i} \mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right]\mathbb{E}\left[\mathbf{u}_{i-k',j}^2\right] \overline{\boldsymbol{K}_{k',j}}\boldsymbol{K}_{k',j} + \overline{\boldsymbol{B}^{(2)}_{i,j}}\boldsymbol{B}^{(2)}_{i,j}\overline{\boldsymbol{W}_{\ell,j}}\mathbb{E}\left[\mathbf{u}_{i,\ell}^2\right]$$

as desired. □