μpscaling Small Models: Principled Warm Starts and Hyperparameter Transfer

Yuxin Ma Nan Chen Mateo Díaz Soufiane Hayou
Dmitriy Kunisky Soledad Villar

Department of Applied Mathematics and Statistics
Johns Hopkins University

Abstract

Modern large-scale neural networks are often trained and released in multiple sizes to accommodate diverse inference budgets. To improve efficiency, recent work has explored *model upscaling*: initializing larger models from trained smaller ones in order to transfer knowledge and accelerate convergence. However, this method can be sensitive to hyperparameters that need to be tuned at the target upscaled model size, which is prohibitively costly to do directly. It remains unclear whether the most common workaround—tuning on smaller models and extrapolating via hyperparameter scaling laws—is still sound when using upscaling. We address this with principled approaches to upscaling with respect to model widths and efficiently tuning hyperparameters in this setting. First, motivated by μP and any-dimensional architectures, we introduce a general upscaling method applicable to a broad range of architectures and optimizers, backed by theory guaranteeing that models are equivalent to their widened versions and allowing for rigorous analysis of infinite-width limits. Second, we extend the theory of μTransfer to a hyperparameter transfer technique for models upscaled using our method and empirically demonstrate that this method is effective on realistic datasets and architectures.

Contents

1 Introduction

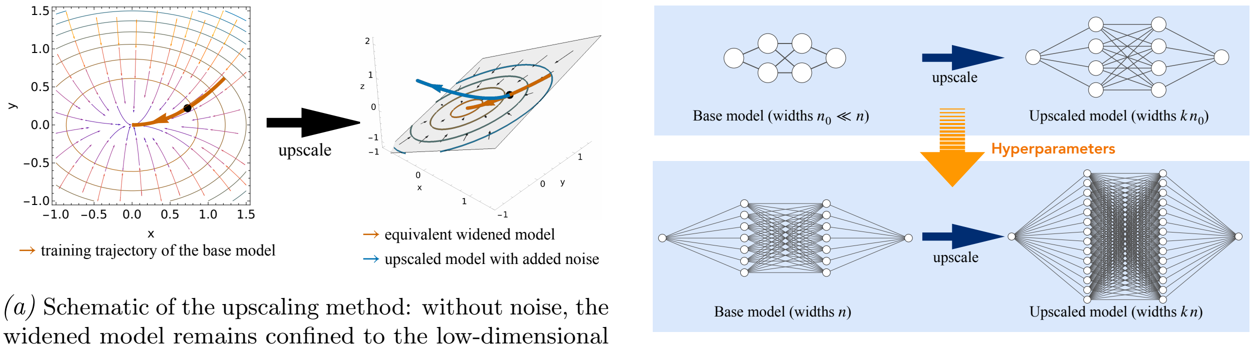
Modern neural network workflows typically involve training families of models at multiple scales. In the early, exploratory stage, smaller models enable rapid prototyping and scaling laws analysis, helping guide architecture choices and forecast the performance of larger variants. Later, for a final release, researchers devote substantially more compute to train much larger models of varying sizes, which are often shipped as suites to meet diverse downstream hardware, latency, and cost constraints (as, e.g., for LLaMA or GPT). As the models of varying scale are typically trained from scratch, this workflow involves a wasteful cycle of disposal and re-training. A natural question is whether smaller trained models can be *upscaled* to warm-start the training of larger ones, transferring information across scales to reduce the total compute required for the final suite.

Several strategies have been proposed (e.g. Chen et al. (2015); Gong et al. (2019); Chen et al. (2022); Kim et al. (2024)), typically employing function-preserving weight transformations that expand a pre-trained smaller model into a larger one so that the larger model produces identical outputs for any input, thereby allowing it to immediately inherit the smaller model's performance before further optimization. However, the training dynamics of upscaled models could be quite different from those of large models trained from scratch. As a result, to use upscaling effectively requires finding good choices of several hyperparameters that are highly dependent on the model architecture and optimizer. In practice, hyperparameter tuning at the scale of modern large models is infeasible, so hyperparameters for training large models are typically chosen by extrapolating from smaller models using scaling laws. Unfortunately, upscaling appears *a priori* to be incompatible with this approach, since training a single model with upscaling involves training *both* at small and then at large scales. Previous work has relied only on informal heuristics for hyperparameter tuning in the presence of upscaling to address this issue.

In this work, we provide a general theoretical framework to "upscale" a pretrained smaller model and its hyperparameters to a larger one, thereby unifying upscaling across different architectures and optimizers in a principled manner.

As our **first contribution**, presented in Section 2, we extend the theory of function-preserving model expansions (Chen et al., 2015). We introduce a procedure that combines weight transformations with a coordinated rescaling of the learning rate and other relevant optimizer hyperparameters, and we prove that the resulting widened model both is functionally identical at initialization and remains equivalent to the original model throughout the entire training trajectory. We further show that this procedure is closely connected to the Maximal Update Parametrization (μP) proposed in Yang and Hu (2020); Yang and Littwin (2023); see Section 2.3. Using this connection, we extend the μP software package to support the widening procedure, facilitating its implementation across standard architectures including multi-layer perceptron (MLP), ResNets, and transformers.

For our **second contribution**, presented in Section 3, we propose an upscaling algorithm grounded in the above theory. Intuitively, this theory implies that the training dynamics of the narrower model evolve within a lower-dimensional subspace of the wider model's ambient parameter space. Therefore, once we widen a model, we also inject a small symmetry-breaking perturbation into the parameters so that the upscaled dynamics can escape this lower-dimensional subspace; see Figure 1a. We leverage the connection with μP to show that our upscaling algorithm—with the right amount of noise—induces training dynamics that are provably stable and exhibit "optimal" feature learning in the infinite-width limit. In the same vein as Yang et al. (2022), our method aligns training dynamics across widths and enables zero-shot hyperparameter transfer: we can now


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0 −0.5 0.0 0.5 1.0 1.5
x
y
training trajectory of the base model
upscale
z
x
y
equivalent widened model
upscaled model with added noise
Base model (widths $n_0 \ll n$)
upscale
Upscaled model (widths $k\,n_0$)
Hyperparameters
Base model (widths n)
upscale
Upscaled model (widths $k\,n$)

(a) Schematic of the upscaling method: without noise, the widened model remains confined to the low-dimensional weight subspace and follows the same trajectory; adding a small perturbation allows it to escape and exploit the additional capacity of the higher-dimensional weight space.

(b) Schematic of the hyperparameter transfer method: transferring hyperparameters from a smaller system of upscaled models to a larger one.

Figure 1. Illustration of upscaling method and hyperparameter transfer method.

easily tune the hyperparameters on narrow models using the same upscaling procedure, and the best hyperparameters found there can then directly be used for large model upscaling; see Figure 1b. Our implementation based on the μP software package also enables straightfoward implementation of hyperparameter transfer.

As our **third contribution**, presented at the end of Section 3.2 with additional details in Appendix E, we use the Tensor Programs machinery to characterize the infinite-width limit of upscaled training for common network architectures, opening the door to further theoretical analysis of training dynamics involving upscaling.

While here we focus on upscaling with respect to "width", extensions to upscaling with respect to depth are a natural next step whose treatment we leave to future work (see for example Yang et al. 2023b; Bordelon et al. 2023 for hyperparameter transfer across depths). In Section 4, we provide experiments showing the performance of the proposed upscaling algorithm across various architectures and datasets, and illustrating the hyperparameter transfer numerically. The related work is summarized in Appendix B, and the conclusions and limitations are given in Appendix G. Our implementation of model upscaling, and the accompanying experiments, is publicly available at `https://github.com/yuxinma98/mupscaling`.

Notation. We use the symbol $\odot$ to denote the Hadamard (entry-wise) product and $\otimes$ to denote the Kronecker product. In particular, if we take a matrix $A \in \mathbb{R}^{m\times n}$ and and the matrix of all ones $\mathbf{1}_k\mathbf{1}_\ell^\top \in \mathbb{R}^{k\times\ell}$, then, $A \otimes (\mathbf{1}_k\mathbf{1}_\ell^\top)$ is a matrix of size $(mk) \times (n\ell)$, that replaces each entry $A_{i,j}$ of A with a $k \times \ell$ block filled with $A_{i,j}$. This operation is depicted in the following diagram

$$\underset{\textbf{Matrix } A\ (m\times n)}{\begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{bmatrix}} \xrightarrow{\otimes \mathbf{1}_k\mathbf{1}_\ell^\top} \overset{\textbf{Expanded Matrix } (mk\times n\ell)}{\begin{bmatrix} k \text{ rows}\left\{\overbrace{\boxed{\begin{matrix} A_{1,1} \\ \text{block}\end{matrix}}}^{\ell \text{ columns}}\right. & \cdots & \boxed{\begin{matrix} A_{1,n} \\ \text{block}\end{matrix}} \\ \vdots & \ddots & \vdots \\ \boxed{\begin{matrix} A_{m,1} \\ \text{block}\end{matrix}} & \cdots & \boxed{\begin{matrix} A_{m,n} \\ \text{block}\end{matrix}} \end{bmatrix}}.$$

2 Equivalent models of different widths

Our goal is to use pre-trained small models as a warm start initialization for training larger models. To do that, we will widen the small model into a larger one that implements the same function so that whatever was learned in the small model is not lost when we train the larger one. In this section, we present a theoretical analysis of the equivalence between neural networks of different widths across a broad range of architectures and optimization methods. We distinguish two notions of equivalence. First, a *static* equivalence refers to two models of different widths parametrizing exactly the same function. Second, a *dynamic* equivalence concerns an architecture–optimizer pair and means that two models of different widths follow identical training trajectories in function space—that is, they parametrize the same function at every training step.

These two perspectives are unified by the framework of tensor programs, which we leverage to obtain general results in Section 2.3. As a warm-up, in Section 2.1 we first illustrate both notions separately using a bias-free MLP trained with vanilla stochastic gradient descent (SGD). The next two subsections generalize this analysis, first to other optimization methods and then to general architectures.

2.1 Warm-up: MLP trained with SGD

Consider an L-layer MLP parameterizing a function from $\mathbb{R}^{d_{\text{in}}}$ to $\mathbb{R}^{d_{\text{out}}}$ that maps $x^{(0)} \mapsto h^{(L)}$ by recursing

$$h^{(\ell)} = W^{(\ell)} x^{(\ell-1)} \in \mathbb{R}^{n_\ell}, \qquad x^{(\ell)} = \phi(h^{(\ell)}) \in \mathbb{R}^{n_\ell}, \qquad \text{for } \ell = 1, 2, \dots, L, \tag{1}$$

where the input and output dimensions are $n_0 = d_{\text{in}}$ and $n_L = d_{\text{out}}$, respectively, $(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}})_{\ell=1}^{L}$ are the layer-wise weight matrices, and ϕ is the activation function applied elementwise at each layer. Fixing n_0 and n_L and increasing the hidden widths $n_1, \dots, n_{L-1}$ yields wider, more expressive MLP models.

We first show that duplicating and appropriately rescaling the weight matrices of any base MLP produces a widened MLP that parameterizes the exact same function.

Proposition 2.1 (Static equivalence of MLPs)**.** *Consider a base MLP with weight matrices* $\big(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}\big)_{\ell=1}^{L}$*. Construct a widened MLP that uses the same activation function and preserves the input and output dimensions, with weights obtained by duplicating and rescaling those of the base MLP as*

$$\big(W^{\uparrow(\ell)} := k_{\ell-1}^{-1}\, W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^{\top} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}\big)_{\ell=1}^{L}, \tag{2}$$

where $N_\ell = k_\ell n_\ell$ for all ℓ and the width multipliers $k_\ell \in \mathbb{N}$ satisfy $k_0 = k_L = 1$. Then, for any input $x^{(0)} \in \mathbb{R}^{d_{\text{in}}}$, both networks produce identical outputs $h^{(L)} \in \mathbb{R}^{d_{\text{out}}}$ and thus parameterize the same function.

Proof. We check by induction over layers that the activations of the widened MLP, denoted by $x^{\uparrow(\ell)} \in \mathbb{R}^{N_\ell}$, are duplicated versions of those of the base MLP, denoted by $x^{(\ell)} \in \mathbb{R}^{n_\ell}$. Specifically, assume $x^{\uparrow(\ell-1)} = x^{(\ell-1)} \otimes \mathbf{1}_{k_{\ell-1}}$. Then

$$\begin{aligned} h^{\uparrow(\ell)} &= W^{\uparrow(\ell)} x^{\uparrow(\ell-1)} \\ &= k_{\ell-1}^{-1}\big(W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^{\top}\big)\big(x^{(\ell-1)} \otimes \mathbf{1}_{k_{\ell-1}}\big) \\ &= h^{(\ell)} \otimes \mathbf{1}_{k_\ell}, \end{aligned}$$

and hence $x^{\uparrow(\ell)} = x^{(\ell)} \otimes \mathbf{1}_{k_\ell}$ as well. □

Following the forward pass, backpropagation for the MLP defined in (1) proceeds as follows:

$$
\begin{aligned}
dh^{(L)} &= \nabla_{h^{(L)}} \mathcal{L} \in \mathbb{R}^{d_{\text{out}}}, \\
dx^{(\ell-1)} &= (W^{(\ell)})^\top dh^{(\ell)} \in \mathbb{R}^{n_{\ell-1}}, \\
dh^{(\ell-1)} &= dx^{(\ell-1)} \odot \phi'(h^{(\ell-1)}) \in \mathbb{R}^{n_{\ell-1}}, \\
dW^{(\ell)} &= dh^{(\ell)} (x^{(\ell-1)})^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad \text{for } \ell = L, L-1, \ldots, 1,
\end{aligned}
\tag{3}
$$

where $\mathcal{L}$ denotes the loss, and each gradient $d\bullet$ equals $\frac{\partial \mathcal{L}}{\partial \bullet}$. Applying SGD with layer-wise learning rate $\gamma^{(\ell)}$ to $W^{(\ell)}$, the weights are updated at each training step t by

$$
W_{t+1}^{(\ell)} := W_t^{(\ell)} - \gamma^{(\ell)} dW_t^{(\ell)},
$$

where $W_t^{(\ell)}, dW_t^{(\ell)}$ denote the weight and the gradient at t-th training step respectively. We further show that if the per-layer learning rates of the widened MLP are chosen as a specific rescaling of those in the base MLP, then the two equivalent models undergo equivalent SGD updates and hence follow identical training trajectories in function space.

Proposition 2.2 (Dynamic equivalence of MLPs trained with SGD)**.** *Suppose we have a base MLP with weights $\left(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}\right)_{\ell=1}^L$ trained by SGD with per-layer learning rates $\gamma^{(\ell)}$. Construct a widened MLP with the same activation function and with weights $\left(W^{\uparrow(\ell)} := k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \in \mathbb{R}^{N_\ell \times N_{\ell-1}}\right)_{\ell=1}^L$, where $N_\ell = k_\ell n_\ell$ for all ℓ and $k_0 = k_L = 1$, and train it by SGD using per-layer learning rates $\gamma^{\uparrow(\ell)} := k_\ell k_{\ell-1}^{-1} \gamma^{(\ell)}$. Then, for all training steps $t \geq 0$, the weights satisfy*

$$
W_t^{\uparrow(\ell)} \;=\; k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \tag{4}
$$

for all $\ell = 1, \ldots, L$, and therefore both networks parametrize the same function at every step (assuming they access the same data and randomness).

Note that if the width multipliers are equal across dimensions ($k_1 = \cdots = k_{L-1}$), then this procedure leaves the learning rates of all hidden weights unchanged ($\gamma^{\uparrow(\ell)} := \gamma^{(\ell)}$ for all $\ell = 2, \ldots, L-1$), and only modify the learning rates for $W^{(1)}$ and $W^{(L)}$.

Proof. By Proposition 2.1, prior to training (at $t = 0$) the two MLPs parametrize the same function. Consequently, the gradient at the output layer matches, $dh^{\uparrow(L)} = dh^{(L)}$, when computed on the same loss and data. One can check by induction that for $\ell = L-1, \ldots, 1$ the backpropagated signals satisfy

$$
dx^{\uparrow(\ell)} = k_\ell^{-1} dx^{(\ell)} \otimes \mathbf{1}_{k_\ell}, \qquad dh^{\uparrow(\ell)} = k_\ell^{-1} dh^{(\ell)} \otimes \mathbf{1}_{k_\ell}, \qquad dW^{\uparrow(\ell)} = k_\ell^{-1} dW^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top.
$$

With these identities, one SGD step on the widened weights yields

$$
\begin{aligned}
W^{\uparrow(\ell)} - \gamma^{\uparrow(\ell)} dW^{\uparrow(\ell)} &= k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top - \left(k_\ell k_{\ell-1}^{-1} \gamma^{(\ell)}\right) \left(k_\ell^{-1} dW^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top\right) \\
&= k_{\ell-1}^{-1} (W^{(\ell)} - \gamma^{(\ell)} dW^{(\ell)}) \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top,
\end{aligned}
$$

which preserves the widening relation (4). By induction over t, (4) holds for all steps. □

2.2 Extension to general optimizers

We proceed to extend the previous observation from vanilla SGD to general entrywise optimizers considered in Yang and Littwin (2023), where parameter updates depend on the current and past gradients. This framework encompasses many commonly used optimizers, including Adam (Kingma, 2014) and AdamW (Reddi et al., 2019).

Definition 2.3 (Entrywise optimizer with weight decay)**.** For a weight matrix $W \in \mathbb{R}^{n\times m}$, an *entrywise optimizer* (with learning rate γ) updates W at training step t according to the following rules for $\alpha \in [n]$ and $\beta \in [m]$.

- Under weight decay with constant λ,

$$(W_{t+1})_{\alpha,\beta} = (W_t)_{\alpha,\beta} - \gamma Q_t\left((dW_0 + \lambda W_0)_{\alpha,\beta}, \dots, (dW_t + \lambda W_t)_{\alpha,\beta}; \varepsilon\right). \tag{5}$$

- Under decoupled weight decay with constant λ,

$$(W_{t+1})_{\alpha,\beta} = (1-\lambda\gamma)(W_t)_{\alpha,\beta} - \gamma Q_t\left((dW_0)_{\alpha,\beta}, \dots, (dW_t)_{\alpha,\beta}; \varepsilon\right). \tag{6}$$

Here, $\varepsilon \in \mathbb{R}^s$ refers to additional hyperparameters that may also be scaled, e.g., `eps` in the PyTorch implementation of Adam (Paszke et al., 2017), and $Q_t : \mathbb{R}^{t+1+s} \to \mathbb{R}$ is an *update function*, acting as a temporal filter of the gradient history, which can encode momentum and adaptivity.

Next, we show that for any such optimizer with a *homogeneous* update function, it is possible to choose the learning rate, weight decay coefficient, and additional hyperparameters so that the widened MLP is dynamically equivalent.

Proposition 2.4 (Dynamic equivalence of MLPs trained with general optimizers)**.** *Consider an entrywise optimizer whose update function Q_t is homogeneous of degree m for all t, i.e, for any $t \in \mathbb{N}$, with $x_0, \dots, x_t \in \mathbb{R}$ and $\varepsilon \in \mathbb{R}^s$, we have that for all $a \in \mathbb{R}$,*

$$Q_t(ax_0, \dots, ax_t; a\varepsilon) = a^m Q_t(x_0, \dots, x_t; \varepsilon).$$

Suppose we have a base MLP with weights $\left(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}\right)_{\ell=1}^L$ trained by the above optimizer with per-layer learning rate $\gamma^{(\ell)}$, weight decay coefficient $\lambda^{(\ell)}$, and additional hyperparemeter $\varepsilon^{(\ell)}$. Construct a widened MLP with the same activation function and with weights $\left(W^{(\ell)\uparrow} := k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top \in \mathbb{R}^{N_\ell \times N_{\ell-1}}\right)_{\ell=1}^L$, where $N_\ell = k_\ell n_\ell$ for all ℓ and $k_0 = k_L = 1$, and train it with the same optimizer using the following hyperparameters:

$$\gamma^{\uparrow(\ell)} := k_\ell^m k_{\ell-1}^{-1}\gamma^{(\ell)}, \qquad \varepsilon^{\uparrow(\ell)} := k_\ell^{-1}\varepsilon^{(\ell)}, \qquad \lambda^{\uparrow(\ell)} := \begin{cases} k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)} & \textit{for vanilla weight decay,} \\ k_{\ell-1}k_\ell^{-m}\lambda^{(\ell)} & \textit{for decoupled weight decay.}\end{cases}$$

Then, for all training steps $t \geq 0$, the weights satisfy

$$W_t^{\uparrow(\ell)} \;=\; k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top$$

for all $\ell = 1, \dots, L$, and therefore both networks parametrize the same function at every step.

We prove the Proposition in Appendix C.1, and describe explicitly how to instantiate it for the SGD (including variants with momentum), Adam, and AdamW optimizers.

2.3 Extension to general network architectures

Finally, we present our general result, showing that the observations made above for MLPs extend to virtually all "standard" neural network architectures. Here, we state an informal result since the formal version requires additional technical background. A detailed version of this result is deferred to Theorem C.5 in Appendix C.2.

Theorem 2.5 (Informal)**.** *Consider a "standard" neural network architecture where the output is multiplied by an additional factor of n^{-1}, replacing the sum readout by a width-normalized mean readout.*[1] *Assume an entrywise optimizer whose update functions are homogeneous of degree m. Suppose that we train a base model with learning rate γ, weight decay coefficient λ, and additional hyperparameter ε. Consider a widened model with the same architecture and depth but larger width. Assume that the widened model's weights are obtained by duplicating units along the designated width axes and rescaling appropriately, and that all hyperparameters are rescaled in tandem according to Table 1. Then, at every training step, the base and widened models parametrize the same function.*

Type of weights	Size growth (base → widened)	Widening operation	Hyperparameters
Scalar-like	width-independent → same	$W^{\uparrow} := W$	$\gamma^{\uparrow} := \gamma, \quad \lambda^{\uparrow} := \lambda, \quad \varepsilon^{\uparrow} := \varepsilon$
Vector-like	$\mathbb{R}^{n \times d} \to \mathbb{R}^{N \times d}$ (width $n \to$ width $N := nk$)	$W^{\uparrow} := W \otimes \mathbf{1}_k$	$\gamma^{\uparrow} := k^m \gamma, \quad \varepsilon^{\uparrow} := k^{-1}\varepsilon$ $\lambda^{\uparrow} := \begin{cases} k^{-1}\lambda & \text{(vanilla)} \\ k^{-m}\lambda & \text{(decoupled)} \end{cases}$
Matrix-like	$\mathbb{R}^{n_{\text{out}} \times n_{\text{in}}} \to \mathbb{R}^{N_{\text{out}} \times N_{\text{in}}}$ (width $n_{\text{out}}, n_{\text{in}} \to$ width $N_{\text{out}} := N_{\text{out}} k_{\text{out}},\ n_{\text{in}} := n_{\text{in}} k_{\text{in}}$)	$W^{\uparrow} := k_{\text{in}}^{-1}\, W \otimes \left(\mathbf{1}_{k_{\text{out}}} \mathbf{1}_{k_{\text{in}}}^{\top}\right)$	$\gamma^{\uparrow} := k_{\text{out}}^{m} k_{\text{in}}^{-1} \gamma, \quad \varepsilon^{\uparrow} := k_{\text{out}}^{-1}\varepsilon,$ $\lambda^{\uparrow} := \begin{cases} k_{\text{in}} k_{\text{out}}^{-1}\lambda & \text{(vanilla)} \\ k_{\text{in}} k_{\text{out}}^{-m}\lambda & \text{(decoupled)} \end{cases}$

Table 1. Construction of weights and optimizer hyperparameters to obtain a widened model with equivalent training trajectories.

The term "standard" architecture refers to any neural network representable in the Ne⊗or⊤ program developed in the Tensor Program literature (Yang, 2019; 2020a; Yang and Littwin, 2023). Intuitively, Ne⊗or⊤ is a formal programming language in which each variable is typed as matrix-like, vector-like, or scalar-like. Given a set of input variables, new variables are generated by the following operations: (i) multiplication of a vector-like variable by a matrix-like one; (ii) elementwise nonlinear transformations of vector-like variables; and (iii) averaging the entries of a vector-like variable. See Appendix C.2.1 for the formal definition. This framework encompasses many widely used architectures and components, including MLPs, RNNs, convolution, attention, pooling layers, skip connections, and batch/layer normalization, as established by Yang (2019). Furthermore, Yang and Littwin (2023) demonstrates that any network specified in the Ne⊗or⊤ program admits a corresponding backpropagation program that is also expressible in Ne⊗or⊤ .

In the Ne⊗or⊤ framework, matrix-like learnable parameters have two dimensions that scale with width, vector-like parameters have one such dimension, and scalar-like parameters do not scale with width. In Table 1, we treat each of these categories separately in describing how to construct a dynamically equivalent widened model. For example, in MLPs, the input and output dimensions are constant, while the hidden layer dimensions scale with width. Consequently, input and output weights as well as hidden biases are classified as vector-like, hidden weights are matrix-like, and the output bias is scalar-like. Similarly, in transformers, the context length remains fixed, while `d_model`,

[1] Standard architectures typically sum along the width axis in the final readout. We instead average. For example, in an MLP where the final readout is $W^{(L)}x^{(L-1)}$ with $W^{(L)} \in \mathbb{R}^{d_{\text{out}} \times n_{L-1}}$, we replace it with $n_{L-1}^{-1} W^{(L)} x^{(L-1)}$.

	Matrix-like	Vector-like	Scalar-like
Multiplier of weight $W = A \cdot \overline{W}$	1	1	1
Init variances $\overline{W}_{\alpha\beta} \sim \mathcal{N}\left(0, B \cdot \overline{\sigma}^2\right)$	n_{in}^{-1}	1	1
Learning rate $\gamma = C \cdot \overline{\gamma}$	$n_{\text{out}}^{m} n_{\text{in}}^{-1}$	n^{m}	1
Weight decay (vanilla) $\lambda = D \cdot \overline{\lambda}$	$n_{\text{in}} n_{\text{out}}^{-1}$	n^{-1}	1
Weight decay (decoupled) $\lambda = \tilde{D} \cdot \overline{\lambda}$	$n_{\text{in}} n_{\text{out}}^{-m}$	n^{-m}	1
Additional hyperparameters $\varepsilon = E \cdot \overline{\varepsilon}$	n_{out}^{-1}	n^{-1}	1
Output	—	n^{-1}	—

Table 2. μP width scalings. The output multiplier in the last row should be interpreted as in in Theorem 2.5. Entries are the width-dependent multiplicative factors $(A, B, C, D, \tilde{D}, E)$ applied to width-independent base constants (denoted with bars). The actual hyperparameters equal the base constants times the listed powers of the widths. See Appendix C.3 for a detailed comparison with the versions presented in Yang et al. (2022); Yang and Littwin (2023).

`n_head`, and `dim_feedforward` could scale with width.[2] Thus, matrix-like parameters in transformers include the query, key, value, and output projection matrices: $W^Q, W^K, W^V, W^O \in \mathbb{R}^{\texttt{d\_model} \times \texttt{d\_model}}$.

This language provides a unified framework for studying static and dynamic equivalence as nearly any relevant neural computation—including forward and backward passes—can be expressed as a Ne⊗or⊤ program. Using this framework, we formally extend Proposition 2.1 and Proposition 2.4 from MLPs to a much broader class of network architectures, via similar inductive reasoning.

Connection to μP. Notably, the widening rules in Table 1 are compatible with the μP scaling of Yang and Hu (2020); Yang and Littwin (2023). For reference, Table 2 summarizes μP, which prescribes width-dependent choices of initialization and optimization hyperparameters so that training dynamics exhibit optimal feature learning behavior in the infinite-width limit. Now, suppose we train the base model under μP and, at some training step, instantiate a widened model by applying the "Widening operation" rules in Table 1. We then continue training the widened model using the same base constants $\overline{\gamma}$, $\overline{\lambda}$, and $\overline{\varepsilon}$ as in the base model. Under μP, the hyperparameters in the widened model automatically rescale to match the "Hyperparameters" column of Table 1. For example, for a matrix-like weight, μP sets the widened-model learning rate to $\gamma^{\uparrow} = N_{\text{out}}^{m} N_{\text{in}}^{-1}\, \overline{\gamma}$, whereas the base model uses $\gamma = n_{\text{out}}^{m} n_{\text{in}}^{-1}\, \overline{\gamma}$; therefore $\gamma^{\uparrow} = k_{\text{out}}^{m} k_{\text{in}}^{-1}\, \gamma$, exactly as required by Table 1. The same reasoning applies to the remaining hyperparameters and to the other weight types. In other words, while the explicit construction of an equivalent widened model may appear involved, under μP the procedure becomes essentially mechanical: one transfers the learned weights according to Table 1, and the associated hyperparameters adjust automatically and match the μP. Consequently, from that point onward, the widened model receives exactly the same parameter updates as the base model, i.e., training proceeds as if no widening had occurred.[3]

In the next section, we leverage this observation and adopt the μP framework throughout

[2] We borrow the notation `d_model`, `n_head`, and `dim_feedforward` from the PyTorch implementation of transformer (Paszke et al., 2017).

[3] To make this statement exact for optimizers that maintain internal state (e.g., momentum or Adam), the optimizer state must also be transferred in a manner consistent with the weight transfer. We defer these implementation details to Appendix D.

the rest of the paper. We show that it yields a simple, easily implementable upscaling algorithm and, moreover, leads to desirable properties: it maintains optimal training dynamics, enables hyperparameter transfer, and facilitates theoretical analysis.

3 Training from upscaled initialization

The previous sections focused on widening procedures that obtain *equivalent* models, retaining the knowledge learned by narrower models. To upscale—i.e., to initialize training of a wider model from an existing narrow model—we apply such a widening procedure and then inject noise to allow further training to exploit the widened model's additional capacity. This raises two practical questions: how much noise to add, and which hyperparameters (particularly the learning rate) to use when training the upscaled model. Naive tuning of these choices can be inefficient. Next, we detail our upscaling algorithm, and then present a principled method for selecting the noise level and learning rate.

3.1 Upscaling algorithm

Meta-algorithm 1 provides pseudo-code for our proposed upscaling method, with additional details deferred to Appendix D. The procedure applies to the general architectures and optimizers discussed in Section 2. For concreteness, Algorithm 2 in Appendix D instantiates the method for an MLP trained with SGD.

Meta-algorithm 1 Upscaling procedure

Input: Base model checkpoint at width n pretrained under μP and the corresponding optimizer checkpoint; expansion multiplier k; noise standard deviation (std) base constant $\overline{\sigma_\Delta}$; learning rate base constant $\overline{\gamma^\uparrow}$.

Output: Trained upscaled model of width $N = nk$.

Step 1. Create an equivalent widened model of width N from the base model's checkpoint using the rules in Table 1.

Step 2. Construct the upscaled model by adding noise to the equivalent widened model in Step 1; the amount of noise obeying the same scaling that μP prescribes for random initialization of a fresh model, and is controlled by the base constant $\overline{\sigma_\Delta}$.

Step 3. Create an optimizer for the upscaled model. Modify its internal state by duplication and rescaling that of the base model's optimizer checkpoint, as done for the weights. The specific rule is given in Appendix D.

Step 4. Train the upscaled model constructed in Step 2 under μP using the optimizer in Step 3, with learning rate base constant $\overline{\gamma^\uparrow}$.

For simplicity, this algorithm assumes that all hidden-layer dimensions are equal (i.e., upscaling from width n to kn), but it extends straightforwardly to heterogeneous hidden-layer widths. Additionally, we highlight an important design choice that will be justified later: the variance of the injected noise is scaled with width in the same way as the initialization variance, as specified in Step 2. According to Theorem 2.5, setting $\overline{\sigma_\Delta} = 0$ (injecting zero noise) and using the same base learning-rate constant as the base model, $\overline{\gamma^\uparrow} = \overline{\gamma}$, yields an equivalent widened model whose training loss and trajectory evolve exactly as if training continued on the base model. In this case, at initialization, the upscaled model's training loss matches the base model's terminal loss, making it substantially better than training the widened model from scratch. But, with zero noise the duplicated weights remain identical throughout training, so the widened model does not exploit its extra capacity at all. Injecting noise by taking $\overline{\sigma_\Delta} > 0$ perturbs the parameters and potentially increases the initialization loss, but breaks the symmetry of the widened model, unlocks its additional capacity, and allows it to represent more complex functions. Empirically, in Section 4 we show that this strategy both accelerates convergence, relative to training from scratch, and achieves comparable or superior terminal loss.

3.2 Choosing hyperparameters for upscaling

The upscaling process introduces new hyperparameters, specifically the noise standard-deviation base constant $\overline{\sigma_\Delta}$ and the learning rate base constant $\overline{\gamma^\uparrow}$. These additional hyperparameters require careful tuning for optimal performance. However, tuning on upscaled models, which typically have extremely large widths, is prohibitively expensive in practice. We now show that zero-shot hyperparameter transfer, similar to the approach in Yang et al. (2022), can be applied to upscaling. Specifically, using our proposed upscaling algorithm, one can efficiently tune these hyperparameters on narrower upscaled models (upscaling from width n_0 to kn_0 for some $n_0 \ll n$) and then directly transfer the selected hyperparameters to the practical setting of upscaling from width n to kn. This procedure significantly reduces the computational overhead associated with hyperparameter tuning.

Why does hyperparameter transfer occur for upscaling? Yang et al. (2022) demonstrates that, for any neural network architecture expressed in the Ne⊗or⊤ program, training under μP yields training dynamics that align across widths, thereby enabling the best hyperparameter tuned on narrow models to be directly transferred to wider ones. At first glance, this result appears inapplicable to upscaling. As we mentioned earlier in Section 2.3, the Ne⊗or⊤ program natively supports only matrix multiplication and elementwise nonlinearities of vectors, and, thus, does not explicitly permit the "duplication" operation $\otimes \mathbf{1}\mathbf{1}^\top$ that upscaling involves. Nevertheless, we provide a workaround showing that the upscaling procedure can be encoded within the Ne⊗or⊤ framework by introducing additional variables, so that the results in Yang et al. (2022) apply directly.

To illustrate the idea, take the MLP example from Section 2.1. Specifically, the first forward propagation for the upscaled network is defined recursively as:

$$\begin{aligned} h^{(\ell)} &= (k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top + \Delta^{(\ell)}) x^{(\ell-1)} \in \mathbb{R}^{N_\ell}, \\ x^{(\ell)} &= \phi(h^{(\ell)}) \in \mathbb{R}^{N_\ell}, \end{aligned}$$

where $\Delta^{(\ell)}$ encodes the Gaussian noise injected into the model after widening. Because of the $\otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top$ operation, these equations do not give a Ne⊗or⊤ program. However, they can be rewritten in terms of variables of the original, not widened, dimension:

$$\begin{aligned} h_{(i)}^{(\ell)} &= \sum_{j=1}^{k_{\ell-1}} \left(k_{\ell-1}^{-1} W^{(\ell)} + \Delta_{(i,j)}^{(\ell)} \right) x_{(j)}^{(\ell-1)} \in \mathbb{R}^{n_\ell}, \\ x_{(i)}^{(\ell)} &= \phi(h_{(i)}^{(\ell)}) \in \mathbb{R}^{n_\ell}, \quad \text{for } i = 1, \ldots, k_\ell, \end{aligned}$$

where $h_{(i)}^{(\ell)} := (h_i^{(\ell)}, h_{i+k_\ell}^{(\ell)}, h_{i+2k_\ell}^{(\ell)}, \ldots) \in \mathbb{R}^{n_\ell}, i \in [k_\ell]$ partition the entries of the widened vector $h^{(\ell)} \in \mathbb{R}^{N_\ell}$, and similarly $(\Delta_{(i,j)}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}})_{i \in [k_\ell], j \in [k_{\ell-1}]}$ partition the entries of the widened matrix $\Delta^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$. The two above sets of equations describe identical dynamics, but the latter is a combination only of matrix multiplications and elementwise nonlinearities, and therefore falls within the Ne⊗or⊤ framework.

We generalize this observation in Appendix E.2, showing that any network architecture expressible in the Ne⊗or⊤ program admits an upscaling procedure that can still be formulated within the Ne⊗or⊤ framework by introducing such partitioned variables. This result justifies the width-dependent parametrization of the injected noise in Step 2 of Meta-Algorithm 1, which is chosen to match that of a fresh initialization under μP. This choice, together with training both the base and upscaled models under μP, ensures that the entire training process (including mid-training upscaling) adheres consistently to μP. Consequently, hyperparameters can be transferred reliably for training procedures including upscaling.

Infinite-width training dynamics. Beyond hyperparameter transfer, employing μP throughout the entire upscaling process confers several practical advantages. Following the analysis in Yang and Hu (2020); Yang et al. (2023a), this parametrization yields "optimal" training dynamics in the infinite-width limit (fix the width multiplier k and let the base width $n \to \infty$). In particular, hidden activations stay at $\Theta(1)$ scale throughout training, reducing the incidence of vanishing or exploding gradients. At the same time, their updates also stay at $\Theta(1)$ scale updates, ensuring the model stays in the rich "feature learning" regime and allowing it to achieve lower loss. For upscaling specifically, this ensures that both the initialization and the injected noise make non-trivial contributions to the widened model's training dynamics, a desirable regime that maintains some signal from the base model while exploiting the benefits of increased width.

Finally, using the Tensor Program framework of Yang and Littwin (2023), we can explicitly characterize the infinite-width limit of the entire training dynamics involving upscaling. To streamline future work, we introduce a modified Tensor Program tailored to upscaled training, which we present in Appendix E.1. Without upscaling, the Tensor Program analysis shows that pre-activations converge to i.i.d. Gaussian random variables in the infinite-width limit. With upscaling by a factor k (i.e., increasing the width from n to kn), pre-activations instead converge to i.i.d. blocks, each a k-dimensional Gaussian random vector that may have non-trivial covariance structure. Our modified framework tracks these vectors explicitly—through their evolving covariance structure—across the entire training trajectory. We illustrate the resulting characterization on two simple MLPs trained with SGD in Appendices E.3 and E.4.

4 Experiments

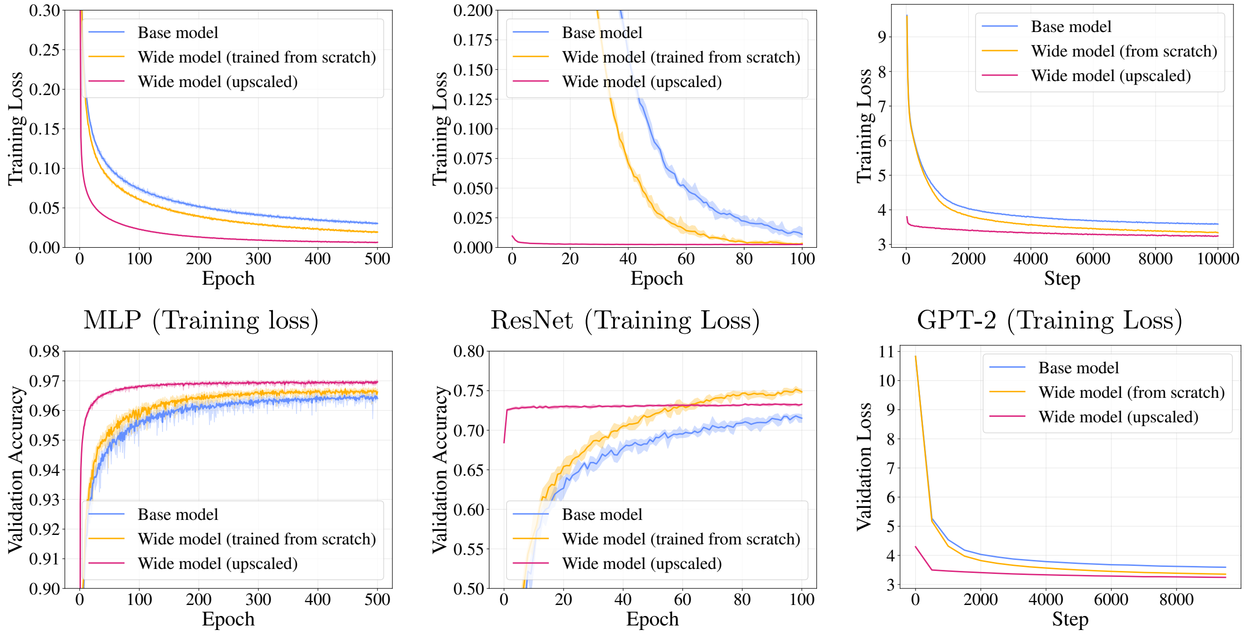
We numerically evaluate the effectiveness of our upscaling algorithm across multiple architectures and optimizers trained on realistic datasets. Figure 2 summarizes results from three representative settings: training an MLP with AdamW on the Forest Cover Type tabular classification dataset (Blackard, 1998), training a ResNet (He et al., 2016) with SGD on the CIFAR-100 image classification benchmark (Krizhevsky et al., 2009), and training GPT-2 (Radford et al., 2019) with AdamW on the FineWeb dataset (Penedo et al., 2024). In all cases, we simulate practical upscaling via the following three-step protocol.

(1) Train a base model of width n from scratch under μP, with hyperparameters either taken from previously reported best settings or obtained via the hyperparameter transfer procedure of Yang et al. (2022) using models of width $n_0 \ll n$.

(2) As a baseline, we train a wider model of width $N = kn$ from scratch under μP, where k is the width multiplier, using the same hyperparameter base constants as in (1), which should remain close to optimal by hyperparameter transfer (Yang et al., 2022).

(3) We then train another wide model of width $N = kn$ using our upscaling algorithm described in Meta-Algorithm 1. For this upscaled model, we tune the std of injected noise and the learning rate on a small upscaled system $n_0 \to kn_0$, while keeping all other hyperparameters identical to those in (1) and (2).

Across experimental settings, upscaled models converge faster and, for a fixed number of training steps, achieve comparable or lower training loss than training from scratch. Extra hyperparameter tuning is performed at the smaller scale kn_0, so it incurs modest cost. On validation, upscaled MLP and GPT-2 mirror these trends, whereas upscaled ResNet generalizes worse than training from scratch, showing an example where upscaling does not help the performance. We also comment that our theory only accounts for the training dynamics, but does not address generalization behavior.

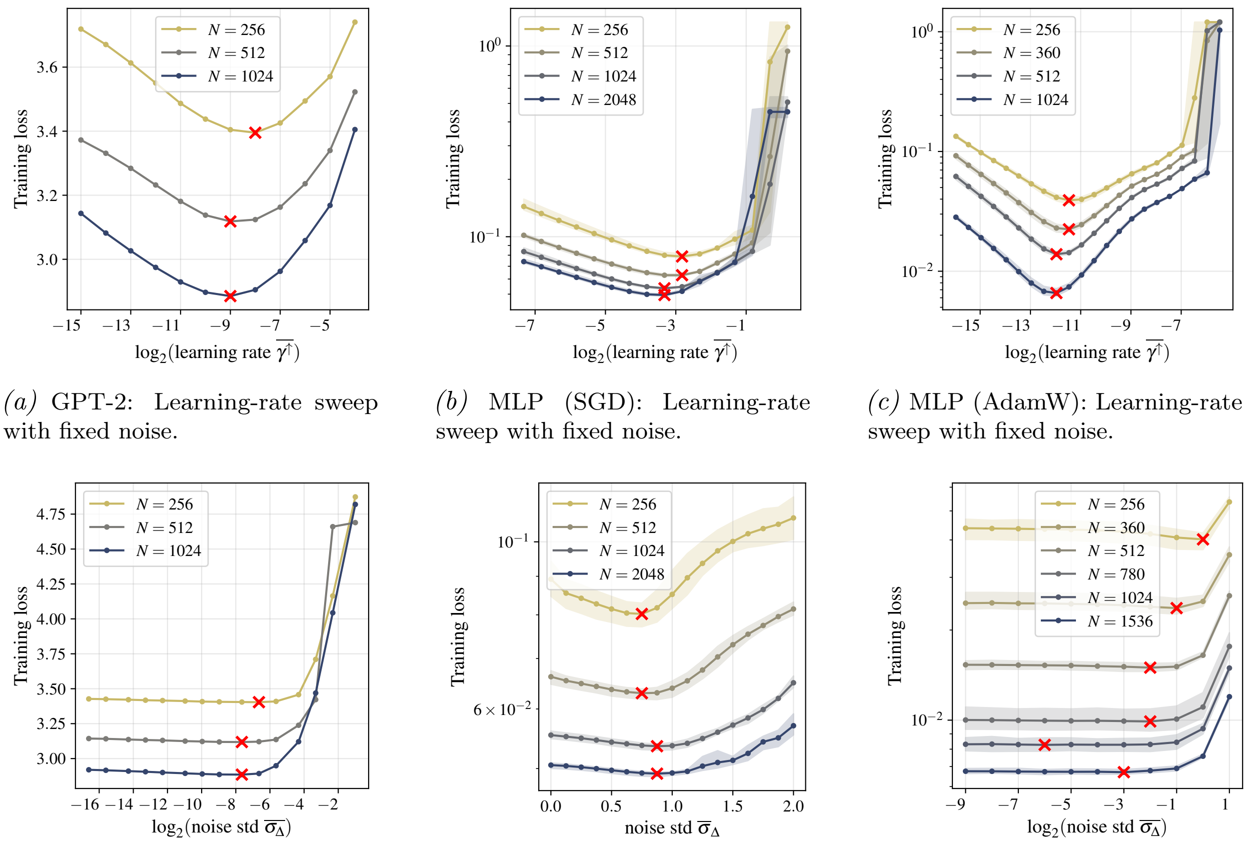


MLP (Training loss) ResNet (Training Loss) GPT-2 (Training Loss)

MLP (Validation accuracy) ResNet (Validation accuracy) GPT-2 (Validation loss)

Figure 2. Training (top row) and validation (bottom row) performance for MLP, ResNet, and GPT-2. The y-axes are truncated to highlight differences between the two curves in each panel. For the MLP and ResNet experiments which have training instability, plots show the mean over five random runs, with shaded min–max bands. More details and additional results are deferred to Appendix F.

Furthermore, we numerically verify that hyperparameter transfer indeed occurs under the upscaling procedure described in Section 3.2. Figure 3(a)(d) reports results for GPT-2, where we train base models with hidden widths $n \in \{128, 256, 512\}$ and upscale each by a multiplier $k = 2$ to widths $N \in \{256, 512, 1024\}$, exploring the injected noise magnitude and the learning rate. In each sweep, we hold one of these hyperparameters fixed while varying the other. Figure 3(b)(e) presents analogous results for MLPs trained with SGD, and Figure 3(c)(f) for MLPs trained with AdamW. In all settings, the optimal hyperparameter choice is stable across widths.



(a) GPT-2: Learning-rate sweep with fixed noise.

(b) MLP (SGD): Learning-rate sweep with fixed noise.

(c) MLP (AdamW): Learning-rate sweep with fixed noise.

(d) GPT-2: Noise sweep with fixed learning rate.

(e) MLP (SGD): Noise sweep with fixed learning rate.

(f) MLP (AdamW): Noise sweep with fixed learning rate.

Figure 3. Hyperparameter transfer for the upscaled model. Columns (left to right): GPT-2 with AdamW, MLP with SGD, and MLP with AdamW. For MLP experiments, curves report the mean across five runs, with min–max ranges across random seeds. In (f), more widths are evaluated than in (c) because of the slightly noisy behavior at $N = 1024$.

Acknowledgments

YM was funded by NSF BSF 2430292 and Amazon AI fellowship. MD was partially supported by NSF awards CCF 2442615 and DMS 2502377. SV was partially funded by NSF CAREER 2339682, NSF CCF 2212457, NSF BSF 2430292, and by the NSF–Simons Research Collaboration on the Mathematical and Scientific Foundations of Deep Learning (MoDL) (NSF DMS 2031985).

References

Jock Blackard. Covertype. UCI Machine Learning Repository, 1998. DOI: https://doi.org/10.24432/C50K5N.

Blake Bordelon, Lorenzo Noci, Mufan Bill Li, Boris Hanin, and Cengiz Pehlevan. Depthwise hyperparameter transfer in residual networks: Dynamics and scaling limit. *arXiv preprint arXiv:2309.16620*, 2023.

Cheng Chen, Yichun Yin, Lifeng Shang, Xin Jiang, Yujia Qin, Fengyu Wang, Zhi Wang, Xiao

Chen, Zhiyuan Liu, and Qun Liu. bert2BERT: Towards reusable pretrained language models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2134–2148, 2022.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Nolan Dey, Bin Claire Zhang, Lorenzo Noci, Mufan Li, Blake Bordelon, Shane Bergsma, Cengiz Pehlevan, Boris Hanin, and Joel Hestness. Don't be lazy: CompleteP enables compute-efficient deep transformers. *arXiv preprint arXiv:2505.01618*, 2025.

Mateo Díaz, Dmitriy Drusvyatskiy, Jack Kendrick, and Rekha R Thomas. Invariant kernels: Rank stabilization and generalization across dimensions. *arXiv preprint arXiv:2502.01886*, 2025.

Wenyu Du, Tongxu Luo, Zihan Qiu, Zeyu Huang, Yikang Shen, Reynold Cheng, Yike Guo, and Jie Fu. Stacking your transformers: A closer look at model growth for efficient LLM pre-training. *Advances in Neural Information Processing Systems*, 37:10491–10540, 2024.

Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of BERT by progressively stacking. In *International conference on machine learning*, pages 2337–2346. PMLR, 2019.

Zhiwei Hao, Jianyuan Guo, Li Shen, Kai Han, Yehui Tang, Han Hu, and Yunhe Wang. Scalenet: Scaling up pretrained neural networks with incremental parameters. *IEEE Transactions on Image Processing*, 34:7109–7122, 2025.

Soufiane Hayou. A proof of learning rate transfer under μP. *arXiv preprint arXiv:2511.01734*, 2025.

Soufiane Hayou and Greg Yang. Width and depth limits commute in residual networks. In *International Conference on Machine Learning*, pages 12700–12723. PMLR, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

Sanghoon Kim, Dahyun Kim, Chanjun Park, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeonwoo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, et al. Solar 10.7 b: Scaling large language models with simple yet effective depth up-scaling. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 23–35, 2024.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.

Eitan Levin and Venkat Chandrasekaran. Free descriptions of convex sets. *arXiv preprint arXiv:2307.04230*, 2023.

Eitan Levin and Venkat Chandrasekaran. Any-dimensional polynomial optimization via de finetti theorems. *arXiv preprint arXiv:2507.15632*, 2025.

Eitan Levin and Mateo Díaz. Any-dimensional equivariant neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 2773–2781. PMLR, 2024.

Eitan Levin, Yuxin Ma, Mateo Díaz, and Soledad Villar. On transferring transferability: Towards a theory for size generalization. *arXiv preprint arXiv:2505.23599*, 2025.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Alexander G de G Matthews, Mark Rowland, Jiri Hron, Richard E Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. *arXiv preprint arXiv:1804.11271*, 2018.

Song Mei, Andrea Montanari, and Phan-Minh Nguyen. A mean field view of the landscape of two-layer neural networks. *Proceedings of the National Academy of Sciences*, 115(33):E7665–E7671, 2018.

Radford M Neal. Bayesian leaning for neural networks, 1996.

Yu Pan, Ye Yuan, Yichun Yin, Zenglin Xu, Lifeng Shang, Xin Jiang, and Qun Liu. Reusing pretrained models by multi-linear operators for efficient training. *Advances in Neural Information Processing Systems*, 36:3248–3262, 2023.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin A Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

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.

Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

Grant Rotskoff and Eric Vanden-Eijnden. Trainability and accuracy of artificial neural networks: An interacting particle system approach. *Communications on Pure and Applied Mathematics*, 75 (9):1889–1935, 2022.

Mohammad Samragh, Iman Mirzadeh, Keivan Alizadeh Vahid, Fartash Faghri, Minsik Cho, Moin Nabi, Devang Naik, and Mehrdad Farajtabar. Scaling smart: Accelerating large language model pre-training with small model initialization. *arXiv preprint arXiv:2409.12903*, 2024.

Peihao Wang, Rameswar Panda, Lucas Torroba Hennigen, Philip Greengard, Leonid Karlinsky, Rogerio Feris, David Daniel Cox, Zhangyang Wang, and Yoon Kim. Learning to grow pretrained models for efficient transformer training. *arXiv preprint arXiv:2303.00980*, 2023.

Greg Yang. Wide feedforward or recurrent neural networks of any architecture are gaussian processes. *Advances in Neural Information Processing Systems*, 32, 2019.

Greg Yang. Tensor programs II: Neural tangent kernel for any architecture. *arXiv preprint arXiv:2006.14548*, 2020a.

Greg Yang. Tensor programs III: Neural matrix laws. *arXiv preprint arXiv:2009.10685*, 2020b.

Greg Yang and Edward J Hu. Feature learning in infinite-width neural networks. *arXiv preprint arXiv:2011.14522*, 2020.

Greg Yang and Etai Littwin. Tensor programs IIb: Architectural universality of neural tangent kernel training dynamics. In *International conference on machine learning*, pages 11762–11772. PMLR, 2021.

Greg Yang and Etai Littwin. Tensor programs IVb: Adaptive optimization in the infinite-width limit. *arXiv preprint arXiv:2308.01814*, 2023.

Greg Yang, Edward J Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs V: Tuning large neural networks via zero-shot hyperparameter transfer. *arXiv preprint arXiv:2203.03466*, 2022.

Greg Yang, James B Simon, and Jeremy Bernstein. A spectral condition for feature learning. *arXiv preprint arXiv:2310.17813*, 2023a.

Greg Yang, Dingli Yu, Chen Zhu, and Soufiane Hayou. Feature learning in infinite-depth neural networks. In *NeurIPS 2023 Workshop on Mathematics of Modern Machine Learning*, 2023b.

Bo-Wen Zhang, Liangdong Wang, Ye Yuan, Jijie Li, Shuhao Gu, Mengdi Zhao, Xinya Wu, Guang Liu, Chengwei Wu, Hanyu Zhao, et al. AquilaMoE: Efficient training for MoE models with scale-up and scale-out strategies. *arXiv preprint arXiv:2408.06567*, 2024.

A Notation

Widening and upscaling. Throughout the paper, n denotes hidden widths. Specifically, n_l denotes the width of the l-th layer of an MLP, while n_{in} and n_{out} denote the input and output dimensions of a weight matrix. We use k to represent the expansion multiplier applied when "widening" or "upscaling" a base model. The hidden widths of the resulting widened or upscaled model are denoted by N, typically satisfying $N = kn$. Quantities associated with the widened or upscaled model (including weights and hyperparameters) are denoted with the superscript $\bullet^{\uparrow}$. We use $\bar{\bullet}$ to denote width-independent quantities, such as the hyperparameter base constants in μP or rescaled variables in "scaled" architectures.

Matrices and vectors. For a vector $x \in \mathbb{R}^{nk}$, we use $x_{(i)} = (x_i, x_{i+k}, \dots) \in \mathbb{R}^n$ for $i \in [k]$ to denote the vectors that partition x. Similarly, for a matrix $W \in \mathbb{R}^{nk \times nk}$, we use $W_{(i,j)} \in \mathbb{R}^{n \times n}$ for $i, j \in [k]$ to denote the blocks that partition W. Layer-specific quantities in an MLP are denoted by superscripts: $W^{(l)}$, $h^{(l)}$, and $x^{(l)}$ represent the weights, pre-activations, and post-activations of the l-th layer, respectively. Subscripts $\bullet_t$ denote quantities at a specific training step t.

Tensor Program. Many of our theoretical results build on the Tensor Program literature, and we follow its notation. For a vector-like quantity x in the Tensor Program, we write $|x\rangle$ for the random variable representing its infinite-width limit, and we decompose it as $|x\rangle = |x\hat{\rangle} + |x\dot{\rangle}$, where $|x\hat{\rangle}$ is the Gaussian part and $|x\dot{\rangle}$ is the correction term. For a scalar-valued quantity c, we denote by $\mathring{c}$ its deterministic limit as the width tends to infinity. Additional notation is introduced in Appendix E. For $x \in \mathbb{R}^n$, we sometimes denote the empirical average by

$$\langle x_\alpha \rangle_\alpha \; = \; \frac{1}{n} \sum_{\alpha \in [n]} x_\alpha .$$

For higher-order tensors $x \in (\mathbb{R}^n)^{\otimes k}$, we write

$$\langle x_{\alpha_1 \dots \alpha_k} \rangle_{\alpha_1 \dots \alpha_k} \; = \; \frac{1}{n^k} \sum_{\alpha_1, \dots, \alpha_k \in [n]} x_{\alpha_1 \dots \alpha_k} .$$

B Related work

Infinite-width limit and hyperparameter transfer. A large body of research seeks to derive tractable descriptions of neural networks in the infinite-width limit, both at initialization and during training. Early work establishes Gaussian-process descriptions at initialization (Neal, 1996; Lee et al., 2017; Matthews et al., 2018). The Neural Tangent Kernel (NTK) characterizes a lazy-training regime in which features remain nearly fixed (Jacot et al., 2018). Complementary analyses based on mean-field theory and PDEs capture aspects of feature learning beyond fixed-kernel approximations (Mei et al., 2018; Rotskoff and Vanden-Eijnden, 2022). More recently, the Tensor Program series (Yang, 2019; 2020a; Yang and Littwin, 2021; Yang, 2020b; Yang and Hu, 2020; Yang and Littwin, 2023; Yang et al., 2022; 2023b;a) provides a unified scaling-limit framework that applies across diverse architectures and optimizers and tracks dynamics in various regimes including the lazy-kernel regime and the feature learning regime. Our work builds on this framework.

The Tensor Program framework captures standard neural computations as compositions of matrix multiplications and elementwise nonlinearities, unified by the $\mathrm{NE}{\otimes}\mathrm{OR}{\top}$ program (see Definition C.3). For neural networks expressible in $\mathrm{NE}{\otimes}\mathrm{OR}{\top}$, and under appropriate scaling, training dynamics in the infinite-width limit can be characterized by deterministic state-evolution recursions that track the distributions of preactivations and model outputs over training. Additional background is provided at the beginning of Appendix E.1. Building on the $\mathrm{NE}{\otimes}\mathrm{OR}{\top}$ scaling-limit analysis, Yang and collaborators introduced the maximal-update parameterization (μP), which prescribes how weights, learning rates, initialization (and related hyperparameters) should scale with width. Within a broad class of width-dependent parameterizations, μP is, in a sense, optimal: it is the unique stable choice under which all parameters are initialized and trained with "maximal" (i.e., non-vanishing, non-exploding) updates in the infinite-width limit. Put differently, μP keeps the network in a feature-learning regime, rather than drifting into a degenerate or purely linearized one. A practical advantage of μP—especially when contrasted with the standard parametrization typically paired with Kaiming initialization (He et al., 2015)—is hyperparameter transfer: hyperparameters tuned on small-width models tend to remain near-optimal as width increases, substantially reducing the cost of scaling up. Extensions of μP and hyperparameter transfer to the infinite-depth setting, as well as to joint infinite-width-and-depth regimes, have been developed and analyzed (Bordelon et al., 2023; Yang et al., 2023b; Hayou and Yang, 2023; Dey et al., 2025). To the best of our knowledge, however, rigorous guarantees of hyperparameter transfer remain scarce: the only formal proof we are aware of is currently limited to linear MLPs (Hayou, 2025).

Model Upscaling. Model upscaling refers to leveraging smaller, pretrained neural networks to initialize the training of larger models within the same architecture family, typically by increasing width and/or depth.[4] The foundational work on Net2Net (Chen et al., 2015) introduced methods for upscaling that preserve the underlying parameterized function. It includes width upscaling (Net2WiderNet), which randomly duplicates weight entries, and depth upscaling (Net2DeeperNet), which inserts identity-mapping layers. It also proposes adding a small amount of noise to break symmetry, as we do in our work. Our upscaling algorithm extends Net2WiderNet (Chen et al., 2015) from function preservation to dynamical equivalence—functional equivalence throughout the training trajectory—by rescaling learning rates and other hyperparameters. This connects our method to μP and the Tensor Program literature on infinite-width analysis and, in turn, enables principled hyperparameter transfer. By contrast, Net2WiderNet provides no tuning guidance: Chen et al. (2015) leaves the symmetry-breaking noise unspecified and reports that optimal learning rates for upscaled models are about 1/10 of those for base models in their experiments. Such shifts necessitate costly retuning, which our framework mitigates through hyperparameter transfer. Additionally, although Chen et al. (2015) demonstrates the method only on MLPs and convolutional networks, we formally establish it for a broad class of architectures. Motivated by the function-preserving philosophy, recent work has empirically explored model upscaling for large language models (LLMs) (Gong et al., 2019; Chen et al., 2022; Samragh et al., 2024; Du et al., 2024; Kim et al., 2024; Zhang et al., 2024) and vision transformers (Hao et al., 2025). Several studies (Chen et al., 2022; Zhang et al., 2024) propose alternative symmetry breaking by mixing parameters from upper layers. In contrast to known function-preserving transformations, (Wang et al., 2023; Pan et al., 2023) develops data-driven upscaling strategies that learn mappings from smaller pretrained models to larger ones, without guarantees on equivalence or training dynamics. Across these approaches, hyperparameter tuning for upscaled models is ad hoc: practitioners either perform costly retuning at the upscaled dimension, reuse base-model hyperparameters (often suboptimal), or apply heuristic adjustments.

Any-dimensional learning. The motivation of our work—particularly the analysis of equivalence across neural-network weight spaces with varying hidden widths in Section 2—originates from a parallel line of research: any-dimensional learning. In our work, we examine the "equivalence" between narrower and wider architectures achieved by expanding a narrow network's weights through "duplication" of entries followed by suitable rescaling. We note a related form of "cross-dimensional equivalence" that we do not pursue here: expanding a narrow model's weights by zero-padding also yields a wider model that is functionally equivalent. This type of "cross-dimensional" equivalence, via duplication or zero padding, has been used to study convex sets (Levin and Chandrasekaran, 2023) and polynomial optimization (Levin and Chandrasekaran, 2025) over varying dimensions. Similar ideas have been used for any-dimensional learning, in which inputs and outputs can be objects of arbitrary size. Architectures operating on sets, graphs, or point clouds are examples of "any-dimensional neural networks:" they use a fixed number of parameters while processing inputs of arbitrary size. Correspondingly, the underlying learning tasks are likewise defined to arbitrary sizes and sometimes require cross-size equivalence via duplication or zero-padding. Along these lines, Levin and Díaz (2024) provides a general framework for constructing equivariant, any-dimensional neural networks; Díaz et al. (2025) explores these ideas in the context of kernel methods; and Levin et al. (2025) investigates size generalization for any-dimensional neural networks. Our work extends this line of research by moving from the input/output space to the weight space, establishing cross-dimensional equivalence among network parameters.

[4] Model upscaling is also known in the literature as knowledge transfer (Chen et al., 2015; Gong et al., 2019; Chen et al., 2022) and model growth (Wang et al., 2023; Pan et al., 2023; Du et al., 2024; Samragh et al., 2024).

C Equivalence between models of different widths

C.1 Missing details from Section 2.2: MLPs with general optimizers

Proof of Proposition 2.4. We proceed by strong induction on the training step t. Suppose $W_s^{\uparrow(\ell)} = k_{\ell-1}^{-1}(W_s^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)$ holds for all $s \leq t$. We will show that it also holds for $t+1$. First, just like in Proposition 2.2, we have

$$dW_s^{\uparrow(\ell)} = k_\ell^{-1} dW_s^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top$$

for all $s \leq t$. We perform different calculations depending on the type of weight decay involved.

Case 1: Vanilla weight decay. Substituting the construction $\gamma^{\uparrow(\ell)} = k_\ell^m k_{\ell-1}^{-1}\gamma^{(\ell)}$, $\lambda^{\uparrow(\ell)} = k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)}$, and $\varepsilon^{\uparrow(\ell)} = k_\ell^{-1}\varepsilon^{(\ell)}$ into the weight update rule (5), we obtain

$$\begin{aligned}
(W^{\uparrow}_{t+1}{}^{(\ell)})_{\alpha,\beta} &= (W_t^{\uparrow(\ell)})_{\alpha,\beta} \\
&\quad - \gamma^{\uparrow(\ell)} Q_t\left((dW_0^{\uparrow(\ell)} + \lambda^{\uparrow(\ell)} W_0^{\uparrow(\ell)})_{\alpha,\beta}, \dots, (dW_t^{\uparrow(\ell)} + \lambda^{\uparrow(\ell)} W_t^{\uparrow(\ell)})_{\alpha,\beta}; \varepsilon^{\uparrow(\ell)}\right) \\
&= (k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta} \\
&\quad - k_\ell^m k_{\ell-1}^{-1}\gamma^{(\ell)} Q_t\Bigg(\left((k_\ell^{-1} dW_0^{(\ell)} + k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)} k_{\ell-1}^{-1} W_0^{(\ell)}) \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top\right)_{\alpha,\beta}, \dots, \\
&\qquad\qquad \left((k_\ell^{-1} dW_t^{(\ell)} + k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)} k_{\ell-1}^{-1} W_t^{(\ell)}) \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top\right)_{\alpha,\beta}; k_\ell^{-1}\varepsilon^{(\ell)}\Bigg).
\end{aligned}$$

By the homogeneity of Q_t (of degree m), this yields

$$\begin{aligned}
&= (k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta} \\
&\quad - k_{\ell-1}^{-1}\gamma^{(\ell)} Q_t\Bigg(\left((dW_0^{(\ell)} + \lambda^{(\ell)} W_0^{(\ell)}) \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top\right)_{\alpha,\beta}, \dots, \\
&\qquad\qquad \left((dW_t^{(\ell)} + \lambda^{(\ell)} W_t^{(\ell)}) \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top\right)_{\alpha,\beta}; \varepsilon^{(\ell)}\Bigg) \\
&= k_{\ell-1}^{-1}(W_{t+1}^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta}.
\end{aligned}$$

Case 2: Decoupled weight decay. Similarly, substituting the scalings $\gamma^{\uparrow(\ell)} = k_\ell^m k_{\ell-1}^{-1}\gamma^{(\ell)}$, $\lambda^{\uparrow(\ell)} = k_{\ell-1}k_\ell^{-m}\lambda^{(\ell)}$, and $\varepsilon^{\uparrow(\ell)} = k_\ell^{-1}\varepsilon^{(\ell)}$ into the weight update rule (6), we obtain

$$\begin{aligned}
(W^{\uparrow}_{t+1}{}^{(\ell)})_{\alpha,\beta} &= \left(1 - \lambda^{\uparrow(\ell)}\gamma^{\uparrow(\ell)}\right)\left((W_t^{\uparrow(\ell)})_{\alpha,\beta} - \gamma^{\uparrow(\ell)} Q_t\left((dW_0^{\uparrow(\ell)})_{\alpha,\beta}, \dots, (dW_t^{\uparrow(\ell)})_{\alpha,\beta}; \varepsilon^{\uparrow(\ell)}\right)\right) \\
&= (1 - \lambda^{(\ell)}\gamma^{(\ell)})\Big((k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta} \\
&\quad - k_\ell^m k_{\ell-1}^{-1}\gamma^{(\ell)} Q_t\Big((k_\ell^{-1} dW_0^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta}, \dots, \\
&\qquad\qquad (k_\ell^{-1} dW_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta}; k_\ell^{-1}\varepsilon^{(\ell)}\Big)\Big).
\end{aligned}$$

Again, by the homogeneity of Q_t (of degree m), this yields

$$= (1 - \lambda^{(\ell)}\gamma^{(\ell)})\Big((k_{\ell-1}^{-1} W_t^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta}$$

$$-k_{\ell-1}^{-1}\gamma^{(\ell)}Q_t\left((dW_0^{(\ell)})_{\alpha,\beta}\otimes\mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top,\dots,(dW_t^{(\ell)})_{\alpha,\beta}\otimes\mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top;\varepsilon^{(\ell)}\right)\Big)$$
$$=k_{\ell-1}^{-1}(W_{t+1}^{(\ell)}\otimes\mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top)_{\alpha,\beta}.$$

The induction is complete in both cases, which establishes the result. □

Next, we instantiate Proposition 2.4 on concrete optimization methods. In particular, we describe how to apply this result to the implementations of these methods found in the PyTorch library.

Example C.1 (SGD with and without momentum)**.** The update function of stochastic gradient descent (SGD) takes the form $Q_t(x_0,\dots,x_t)=x_t$, i.e., a degree-one homogeneous map. Hence, one should choose learning rate $\gamma^{\uparrow(\ell)}:=k_\ell k_{\ell-1}^{-1}\gamma^{(\ell)}$ to ensure equivalent updates between equivalent weights. For SGD with momentum β and dampening τ, the update function is given by

$$Q_t(x_0,\dots,x_t)=(1-\tau)\sum_{s=0}^{t}\beta^{t-s}x_s.$$

This function is again degree-one homogeneous. Hence, using the same learning rate $\gamma^{\uparrow(\ell)}$, we also obtain dynamic equivalence.Further, for SGD with Nesterov momentum, i.e., (`nesterov=True` in `PyTorch`), the update function is given by

$$Q_t(x_0,\dots,x_t)=(1+\beta-\beta\tau)x_t+(1-\tau)\sum_{s=0}^{t-1}\beta^{t-s+1}x_s,$$

which once more is degree-one homogeneous. So, the same conclusion applied. Finally, in `PyTorch` implementation of SGD, the weight decay is not implemented in a decoupled way as in Definition 2.3, so one should choose a weight decay constant for the widened model of

$$\lambda^{\uparrow(\ell)}:=k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)}.$$

Example C.2 (Adam and AdamW)**.** The Adam optimizer (Kingma, 2014) with hyperparameters β_1,β_2 is also an entrywise optimizer with update function

$$Q_t(x_0,\dots,x_t;\varepsilon)=\frac{(1-\beta_1^t)^{-1}(1-\beta_1)\sum_{s=0}^t\beta_1^{t-s}x_s}{\sqrt{(1-\beta_2^t)^{-1}(1-\beta_2)\sum_{s=0}^t\beta_2^{t-s}x_s^2}+\varepsilon},$$

where $\varepsilon>0$ is a small constant for numerical stability. The function Q_t is a degree-zero homogeneous map. Hence, for Adam, one should use the following hyperparameters for the widened model

$$\gamma^{\uparrow(\ell)}:=k_{\ell-1}^{-1}\gamma^{(\ell)},\quad \varepsilon^{\uparrow(\ell)}:=k_\ell^{-1}\varepsilon^{(\ell)}.$$

The update function of AMSGrad (Reddi et al., 2019) (`amsgrad=True` in `PyTorch`) is given by

$$Q_t(x_0,\dots,x_t;\varepsilon)=\frac{(1-\beta_1^t)^{-1}(1-\beta_1)\sum_{s=0}^t\beta_1^{t-s}x_s}{\sqrt{\max\left(v_0(x_0),\dots,v_t(x_0,\dots,x_t)\right)}+\varepsilon},$$
$$v_t(x_0,\dots,x_t)=(1-\beta_2^t)^{-1}(1-\beta_2)\sum_{s=0}^t\beta_2^{t-s}x_s^2,$$

which is again degree-zero homogeneous. Hence, the same choice applies here. Once more, by default, the weight decay is not implemented in a decoupled way in the Adam optimizer in `PyTorch`, so one should choose

$$\lambda^{\uparrow(\ell)} := k_{\ell-1}k_\ell^{-1}\lambda^{(\ell)}.$$

Finally, for the AdamW optimizer, i.e., `decoupled_weight_decay=True` in `Adam`, weight decay *is* implemented in a decoupled way (Loshchilov and Hutter, 2017). Hence, one should instead use

$$\lambda^{\uparrow(\ell)} := k_{\ell-1}\gamma^{(\ell)}.$$

C.2 Missing details from Section 2.3: equivalence for general network architectures

To extend these results to all "standard" architectures, we will use Ne⊗or⊤ programs. Appendix C.2.1 starts by reviewing the version of these programs appearing in Yang and Littwin (2023), which is more general than those in earlier papers on the subject. With this background in place, we state a formal version of Theorem 2.5 in Theorem C.5. We then present a slightly more general formulation of the same result in Appendix C.2.2, followed by a proof in Appendix C.2.3.

C.2.1 Background: Ne⊗or⊤ program and the backpropagation program

We briefly review the Ne⊗or⊤ language (Definition 2.6.1 in Yang and Littwin (2023)). We introduce a minor extension that allows "width" dimensions to vary across layers. In the original Ne⊗or⊤ construction used for infinite-width analysis, assuming equal layer widths was essentially a notational convenience: because all widths are taken to diverge at the same rate, any finite discrepancies between them vanish asymptotically and can be ignored. For our purpose, we explicitly allow varying widths.

Definition C.3 (Ne⊗or⊤ program)**.** A Ne⊗or⊤ program is an iterative procedure that generates a sequence of vectors $\boldsymbol{x}$ and a sequence of real scalars $\boldsymbol{c}$, defined inductively from an initial collection of scalars $\boldsymbol{c}^0 \subseteq \boldsymbol{c}$, an initial collection of vectors $\boldsymbol{x}^0 \subseteq \boldsymbol{x}$, and an initial set of matrices $\mathcal{W}$, by repeatedly applying any of the following allowed operations.[5]

- `Avg`: Choose a vector $x \in \boldsymbol{x}$ of dimension n, and append to $\boldsymbol{c}$ a scalar

$$\langle x_\alpha \rangle_\alpha = \frac{1}{n}\sum_{\alpha=1}^{n} x_\alpha \in \mathbb{R}.$$

- `MatMul`: Choose a matrix $W \in \mathcal{W}$ and vector $x \in \boldsymbol{x}$ with compatible dimensions, and append to $\boldsymbol{x}$ the vector

$$Wx, \quad \text{or} \quad W^\top x. \tag{7}$$

- `OuterNonlin`: Given integers $r \geq 0, n \in \mathbb{N}$ and a function $\psi : \mathbb{R}^{|\tilde{\boldsymbol{x}}|(r+1)+l} \to \mathbb{R}$, append to $\boldsymbol{x}$ the vector

$$y \in \mathbb{R}^n, \quad y_\alpha = \langle \psi(\tilde{\boldsymbol{x}}_\alpha; \tilde{\boldsymbol{x}}_{\beta_1}; \ldots; \tilde{\boldsymbol{x}}_{\beta_r}; \boldsymbol{c})\rangle_{\beta_1,\ldots,\beta_r} = \frac{1}{n^r}\sum_{\beta_1,\ldots,\beta_r=1}^{n} \psi(\tilde{\boldsymbol{x}}_\alpha; \tilde{\boldsymbol{x}}_{\beta_1}; \ldots; \tilde{\boldsymbol{x}}_{\beta_r}; \boldsymbol{c}).$$

[5] For this section, we slightly depart from Definition 2.6.1 of Yang and Littwin (2023): The initial scalars, vectors, and matrices are taken to be deterministic, rather than randomly initialized under the prescribed rules.

Here, $\tilde{\boldsymbol{x}} \subseteq \boldsymbol{x}$ denotes the subset of the vectors with the same dimension n, and we will think of $\tilde{\boldsymbol{x}}$ as a matrix with n-dimensional columns. We write $\tilde{\boldsymbol{x}}_\gamma$ for the γth row of the matrix $\tilde{\boldsymbol{x}}$, and $|\tilde{\boldsymbol{x}}|$ denotes the number of columns in the matrix $\tilde{\boldsymbol{x}}$.

We emphasize that a NE⊗ORT program itself is merely the syntactic structure of the above transformations, not their evaluations on any particular family of scalars, vectors, and matrices. It may be viewed as specified by an abstract syntax tree, for instance, with nodes representing the above operations (together with the nonlinearities ψ).

Consider a neural network architecture parameterizing a function $\mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ with a weight space consisting of ℓ matrices, m vectors, and j scalar weights. Suppose the matrices have dimensions $n_{1,\text{out}} \times n_{1,\text{in}}, \ldots, n_{\ell,\text{out}} \times n_{\ell,\text{in}}$, and the vectors have dimensions dimensions $n_1, \ldots, n_m$. We denote the weight space as

$$\mathcal{T}_{\boldsymbol{n}} = (\mathbb{R})^j \oplus \left(\bigoplus_{i=1}^{m} \mathbb{R}^{n_i} \right) \oplus \left(\bigoplus_{i=1}^{\ell} \mathbb{R}^{n_{i,\text{out}} \times n_{i,\text{in}}} \right),$$

which is indexed by the "width vector" that collects the hidden widths of the weights.

$$\boldsymbol{n} = (n_1, \ldots, n_m, n_{1,\text{in}}, n_{1,\text{out}}, \ldots, n_{\ell,\text{in}}, n_{\ell,\text{out}}).$$

A NE⊗ORT program π *represents* this neural network architecture if the following properties hold. First, π starts with the initial set of scalars $\boldsymbol{c}_0$ consisting of the (initialized) j scalar weights and the input of dimension d_{in}, the initial set of vectors $\boldsymbol{x}_0$ consisting of the (initialized) m vector weights, and the initial set of matrices $\mathcal{W}$ consisting of the (initialized) ℓ matrix weights in $\mathcal{T}_{\boldsymbol{n}}$. The NE⊗ORT program then describes the computations of all intermediate values in the architecture's forward pass. At the end, it picks vectors $(x^1, \ldots, x^{d_{\text{out}}})$ from the final set of vectors $\boldsymbol{x}$, and outputs $y \in \mathbb{R}^{d_{\text{out}}}$ with entries

$$y_i = \sum_{\alpha} x^i_\alpha, \quad i = 1, \ldots, d_{\text{out}}. \tag{8}$$

Appendix A of Yang (2019) shows that many common neural network components—BatchNorm, skip connections, convolution, pooling, GRU, LSTM, layer normalization, and scaled attention—are expressible in NE⊗ORT , and by composing these we see that NE⊗ORT programs can represent standard CNN, RNN, and Transformer architectures.

Given a program π representing an architecture, Definition 2.9.14 of Yang and Littwin (2023) shows that one can automatically construct another NE⊗ORT program for backpropagation to compute all of the gradient vectors with respect to x needed to perform gradient updates. A few times here and in the discussion below, we will talk about automatically building a new NE⊗ORT program from a given one. Since we view NE⊗ORT as a formal programming language, this kind of procedure should be thought of as akin to compilation of ordinary computer programs: we perform automated transformations on a NE⊗ORT program to turn it into another NE⊗ORT program, perhaps having different operational semantics.

Definition C.4 (Backpropagation program)**.** Consider any NE⊗ORT program π and a vector $x \in \mathbb{R}^{n_x}$ in π. Then π's *backpropagation program* with respect to x is an extension of π defined by constructing the following objects on top of π: (Intuitively, one should interpret $d^x y = n_y \frac{\partial \langle x_\alpha \rangle_\alpha}{\partial y}$ if $y \in \mathbb{R}^{n_y}$ is a vector and $d^x c = \frac{\partial \langle x_\alpha \rangle_\alpha}{\partial c}$ if c is a scalar.)

- $d^x x := \mathbf{1}_{n_x} \in \mathbb{R}^{n_x}$.
- For any `MatMul` instruction $z := Wy$ in π, we construct

$$d^{x|z} y := W^\top d^x z \text{ (via another } \texttt{MatMul}\text{)}. \tag{9}$$

- For any `Avg` instruction $c := \langle x_\alpha \rangle_\alpha$ in π, we construct

  $$d^{x|c}z := (d^x c)\mathbf{1}_{n_z} \in \mathbb{R}^{n_z} \quad \text{(via } \texttt{OuterNonlin}\text{)}.$$

- For any `OuterNonlin` instruction $y := \langle \psi(\tilde{\boldsymbol{x}}; \tilde{\boldsymbol{x}}_{\beta_1}, \ldots, \tilde{\boldsymbol{x}}_{\beta_r}; \boldsymbol{c}) \rangle_{\beta_1, \ldots, \beta_r}$, for each $i = 0, \ldots, r$, let

  $$\mathbf{g}^i_{\beta_0 \ldots \beta_r} := d^x y \psi_i(\tilde{\boldsymbol{x}}_{\beta_0}, \ldots, \tilde{\boldsymbol{x}}_{\beta_r}; \boldsymbol{c}) \in \mathbb{R}^{|\tilde{\boldsymbol{x}}|},$$

  where $\psi_i : \mathbb{R}^{|\tilde{\boldsymbol{x}}|(r+1)+\ell} \to \mathbb{R}^{|\tilde{\boldsymbol{x}}|}$ yields the derivative of ψ with respect to x in the i-th slot. When $i = r+1$, we make the analogous definition for $\mathbf{g}^{r+1}_{\beta_0 \ldots \beta_r} \in \mathbb{R}^{|\boldsymbol{c}|}$. We write $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_r)$, $\boldsymbol{\beta}[i \mapsto \alpha] = (\beta_0, \ldots, \beta_{i-1}, \alpha, \beta_{i+1}, \ldots, \beta_r)$, and $\boldsymbol{\beta}_{-i} = (\beta_0, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_r)$. Then we construct $d^{x|y}c = (d^{x|y}c^1, \ldots, d^{x|y}c^{|c|})$ and $d^{x|y}\boldsymbol{x} = (d^{x|y}x^1, \ldots, d^{x|y}x^{|\boldsymbol{x}|})$ by

  $$d^{x|y}\boldsymbol{c} := \langle \mathbf{g}^{r+1}_{\boldsymbol{\beta}} \rangle_{\boldsymbol{\beta}} \in \mathbb{R}^{|\boldsymbol{c}|} \quad \text{(using } \texttt{OuterNonlin} \text{ and } \texttt{Avg}\text{)}$$

  $$d^{x|y}\boldsymbol{x}_\alpha := \sum_{i=0}^{r} \langle \mathbf{g}^i_{\boldsymbol{\beta}[i \mapsto \alpha]} \rangle_{\boldsymbol{\beta}_{-i}} \in \mathbb{R}^{|\boldsymbol{x}|} \quad \text{(using } \texttt{OuterNonlin}\text{)}.$$

  Explicitly,

  $$d^{x|y}\boldsymbol{x}_\alpha = \langle \mathbf{g}^0_{\alpha\beta_1 \ldots \beta_r} \rangle_{\beta_1 \ldots \beta_r} + \langle \mathbf{g}^1_{\beta_0 \alpha \beta_2 \ldots \beta_r} \rangle_{\beta_0 \beta_2 \ldots \beta_r} + \cdots + \langle \mathbf{g}^r_{\beta_0 \ldots \beta_{r-1}\alpha} \rangle_{\beta_0 \ldots \beta_{r-1}}.$$

- Finally, for every vector or scalar y in π other than x,

  $$d^x y := \sum_u d^{x|u} y$$

  where u ranges over all vectors or scalars in π whose construction used y.

The subprogram that constructs all of these new objects is denoted $d^x\pi$. For a NE⊗OR⊤ program π representing a neural-network architecture, backpropagation involves the subprograms $d^{x^1}\pi, \ldots, d^{x^{d_{\text{out}}}}\pi$, where $(x^1, \ldots, x^{d_{\text{out}}})$ are vectors used to generate the output in (8).

With this background, we now write the formal version of Theorem 2.5, that we will proceed to prove in the rest of this section.

Theorem C.5 (Formal version of Theorem 2.5)**.** *Consider a neural network architecture represented by a* NE⊗OR⊤ *program π satisfying the following two rules.*

1. *Its forward pass only uses matrix multiplication of the form Wx, not of the form $W^\top x$.*

2. *Its readout rule (8) replaces the sum by the average, i.e., $y_i = \langle x^i_\alpha \rangle_\alpha$.*

Assume an entrywise optimizer whose update function is homogeneous of degree m. Consider a base model under this architecture, with widths $\boldsymbol{n} = (n_1, \ldots, n_m, n_{1,\text{in}}, n_{1,\text{out}}, \ldots, n_{\ell,\text{in}}, n_{\ell,\text{out}})$ and initial weights

$$\Theta = \big(\underbrace{\theta_1, \ldots, \theta_j}_{\textit{scalars}}, \underbrace{x_1, \ldots, x_m}_{\textit{vectors}}, \underbrace{W_1, \ldots, W_\ell}_{\textit{matrices}}\big) \in \mathcal{T}_{\boldsymbol{n}}.$$

It is trained with the entrywise optimizer with learning rate γ, weight decay constant λ, and additional hyperparameter ε.

Construct a widended model under the same architecture with widths

$$\boldsymbol{N} = \boldsymbol{k} \odot \boldsymbol{n} = (k_1 n_1, \ldots, k_m n_m, k_{1,\text{in}} n_{1,\text{in}}, k_{1,\text{out}} n_{1,\text{out}}, \ldots, k_{\ell,\text{in}} n_{\ell,\text{in}}, k_{\ell,\text{out}} n_{\ell,\text{out}}),$$

where the width multipliers are summarized in the vector $\boldsymbol{k} = (k_1, \ldots, k_m, k_{1,\text{in}}, k_{1,\text{out}}, \ldots, k_{\ell,\text{in}}, k_{\ell,\text{out}})$*. Its weights* $\Theta^{\uparrow} \in \mathcal{T}_{\boldsymbol{n}}$ *are constructed using the rule in the "Widening operation" column of Table 1. That is,*

$$\Theta^{\uparrow} := \big(\underbrace{\theta_1, \ldots, \theta_j}_{\textit{scalars}}, \underbrace{x_1 \otimes \mathbf{1}_{k_1}, \ldots, x_m \otimes \mathbf{1}_{k_m}}_{\textit{vectors}}, \underbrace{k_{1,\text{in}}^{-1} W_1 \otimes \mathbf{1}_{k_{1,\text{out}}} \mathbf{1}_{k_{1,\text{in}}}^{\top}, \ldots, k_{\ell,\text{in}}^{-1} W_\ell \otimes \mathbf{1}_{k_{\ell,\text{out}}} \mathbf{1}_{k_{\ell,\text{in}}}^{\top}}_{\textit{matrices}}\big). \quad (10)$$

The widened model is trained with the same optimizer using per-weight hyperparameters given by the "Hyperparameters" column of Table 1.

Then, at all training steps, both models parametrize the same function.

C.2.2 Formulation with scaled architecture

Theorem C.5 does not apply to architectures whose forward pass simultaneously uses both Wx and $W^\top x$. This restriction is typically harmless: allowing both W and $W^\top$ in the definition of the Ne⊗or⊤ program was intended to capture backpropagation. (If $y = Wx$ appears in the forward pass, then $dx = W^\top dy$ naturally appears in backpropagation.) Standard architectures do not employ $W^\top$ in the forward pass. However, extending the result to architectures that include both operations is straightforward: for the Wx operation use $k_{\text{in}}^{-1} W \otimes \mathbf{1}\mathbf{1}^\top$, and for the $W^\top x$ operation use $k_{\text{out}}^{-1} W^\top \otimes \mathbf{1}\mathbf{1}^\top$, i.e., adopt distinct scaling rules for W and $W^\top$. We will formally state the result in this more general setting (see Theorem C.8) because it contains insights of independent interest.

We begin with observing an equivalent formulation of Proposition 2.4 for MLPs. This formulation rescales the MLP weights so that constructing an equivalent MLP is width-independent—duplicate the weights with no further rescaling.

Define a "scaled" MLP that maps $x^{(0)}$ to $x^{(L)}$ via the recursion

$$\begin{aligned} h^{(\ell)} &= n_{\ell-1}^{-1} W^{(\ell)} x^{(\ell-1)} \in \mathbb{R}^{n_\ell}, \\ x^{(\ell)} &= \phi(h^{(\ell)}) \in \mathbb{R}^{n_\ell}, \quad \text{for } \ell = 1, 2, \ldots, L. \end{aligned}$$

We train the weights $n_{\ell-1}^{-1} W^{(\ell)}$ using the entrywise optimizer, where we adopt μP layer-wise hyperparameters when updating each $n_{\ell-1}^{-1} W^{(\ell)}$ as in Table 2, i.e.,

$$\begin{aligned} \gamma^{(\ell)} &:= n_\ell^m n_{\ell-1}^{-1} \overline{\gamma}, \\ \varepsilon^{(\ell)} &:= n_\ell^{-1} \overline{\varepsilon}, \\ \lambda^{(\ell)} &:= \begin{cases} n_{\ell-1} n_\ell^{-1} \overline{\lambda} & \text{for vanilla weight decay,} \\ n_{\ell-1} n_\ell^{-m} \overline{\lambda} & \text{for decoupled weight decay.} \end{cases} \end{aligned}$$

Here $\overline{\gamma}, \overline{\varepsilon}, \overline{\lambda}$ denote the base learning rate, base auxiliary hyperparameter, and base weight decay constant, respectively, and the actual hyperparameters are scaled according to the layer widths.

Corollary C.6 (Alternative statement of Proposition 2.4)**.** *Consider a base "scaled" MLP with weight matrices* $\big(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}\big)_{\ell=1}^{L}$*. Construct a widened "scaled" MLP that uses the same activation function and preserves the input and output dimensions, with weights obtained by duplicating those of the base MLP as*

$$W^{\uparrow(\ell)} := W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^{\top} \in \mathbb{R}^{N_\ell \times N_{\ell-1}} \quad \textit{for } \ell = 1, \ldots, L.$$

Here $N_\ell = k_\ell n_\ell$ for all ℓ, and the width multipliers $k_\ell \in \mathbb{N}$ satisfy $k_0 = k_L = 1$. Suppose both models are trained with the same base hyperparameter constants $\overline{\gamma}, \overline{\varepsilon}, \overline{\lambda}$ using the entrywise optimizer, with layer-wise hyperparameters scaled accordingly. Then, at all training steps, both models parameterize the same function.

Proof. By Proposition 2.4, the widened "scaled" MLP parameterizes the same function at each training step if the following holds:

$$\begin{aligned}
N_{\ell-1}^{-1} W^{\uparrow(\ell)} &= k_{\ell-1}^{-1}\big(n_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top\big),\\
(N_\ell^m N_{\ell-1}^{-1})\overline{\gamma} &= k_\ell^m k_{\ell-1}^{-1}(n_\ell^m n_{\ell-1}^{-1})\overline{\gamma},\\
(N_\ell^{-1})\overline{\varepsilon} &= k_\ell^{-1}(n_\ell^{-1})\overline{\varepsilon},\\
(N_{\ell-1} N_\ell^{-1})\overline{\lambda} &= k_{\ell-1}k_\ell^{-1}(n_{\ell-1}n_\ell^{-1})\overline{\lambda} \quad \text{for vanilla weight decay,}\\
(N_{\ell-1} N_\ell^{-m})\overline{\lambda} &= k_{\ell-1}k_\ell^{-m}(n_{\ell-1}n_\ell^{-m})\overline{\lambda} \quad \text{for decoupled weight decay.}
\end{aligned}$$

These equalities do hold exactly, establishing the claim. □

To extend these results to general architectures, we first define a "scaled" architecture in which weights are rescaled with respect to width.

Definition C.7 (Scaled architecture)**.** A *scaled* N<span style="font-variant:small-caps">e</span>⊗<span style="font-variant:small-caps">or</span>⊤ *program* is the same as the NE⊗OR⊤ program in Definition C.3, with the modification that in each `MatMul` operation (7) involving $W \in \mathbb{R}^{n_{\text{out}}\times n_{\text{in}}}$,

$$Wx \;\text{ is replaced by }\; \frac{1}{n_{\text{in}}}Wx, \qquad \text{and} \qquad W^\top x \;\text{ is replaced by }\; \frac{1}{n_{\text{out}}}W^\top x.$$

A *scaled neural network architecture* represented by a scaled NE⊗OR⊤ program π is defined in the same way as before, except that at the final readout step (8), the output $y \in \mathbb{R}^{d_{\text{out}}}$ is given by

$$y_i \;= \langle x_\alpha^i \rangle_\alpha = \frac{1}{n^{(i)}} \sum_{\alpha=1}^{n^{(i)}} x_\alpha^i, \qquad i = 1, \dots, d_{\text{out}}, \tag{11}$$

where each chosen vector $x^i \in \mathbb{R}^{n^{(i)}}$. The modification is that we replace the sum with the mean. Because the scaled architecture can be instantiated at multiple hidden widths, it parameterizes a family of functions

$$f_{\boldsymbol{n}} \colon \mathbb{R}^{d_{\text{in}}} \times \mathcal{T}_{\boldsymbol{n}} \to \mathbb{R}^{d_{\text{out}}},$$

indexed by the hidden-width vector $\boldsymbol{n} = (n_1, \dots, n_m, n_{1,\text{in}}, n_{1,\text{out}}, \dots, n_{\ell,\text{in}}, n_{\ell,\text{out}})$.

With this modification, the *scaled backpropagation program* is adjusted accordingly from Definition C.4. For any `MatMul` instruction $z := n_{\text{in}}^{-1}Wy$ in π, the construction of the gradient (9) is replaced by $d^{x|z}y := n_{\text{out}}^{-1}W^\top d^x z$ (via another `MatMul`). Notice that the modified backpropagation program is then a scaled NE⊗OR⊤ program.

With this definition in place, we state the general result that we will prove. One can verify that Theorem C.5 is a special case of the following theorem in which the forward pass does not use both W and $W^\top$.

Theorem C.8. *Consider a scaled architecture represented by a scaled* NE⊗OR⊤ *program π, and an entrywise optimizer whose update function is homogeneous of degree m. Adopt the μP scaling for* *its hyperparameters as in Table 2. Consider a base model under this scaled architecture with widths* $\boldsymbol{n} = (n_1, \dots, n_m, n_{1,\text{in}}, n_{1,\text{out}}, \dots, n_{\ell,\text{in}}, n_{\ell,\text{out}})$ *and initial weights*

$$\Theta = \big(\underbrace{\theta_1, \dots, \theta_j}_{\textit{scalars}}, \underbrace{x_1, \dots, x_m}_{\textit{vectors}}, \underbrace{W_1, \dots, W_\ell}_{\textit{matrices}}\big) \in \mathcal{T}_{\boldsymbol{n}}.$$

Construct a widened model under the same scaled architecture with widths

$$\boldsymbol{N} = \boldsymbol{k} \odot \boldsymbol{n} = (k_1 n_1, \dots, k_m n_m, k_{1,\text{in}} n_{1,\text{in}}, k_{1,\text{out}} n_{1,\text{out}}, \dots, k_{\ell,\text{in}} n_{\ell,\text{in}}, k_{\ell,\text{out}} n_{\ell,\text{out}}),$$

where the width multipliers are summarized in the vector $k = (k_1, \dots, k_m, k_{1,\text{in}}, k_{1,\text{out}}, \dots, k_{\ell,\text{in}}, k_{\ell,\text{out}})$*. Its weights* $\Theta^\uparrow \in \mathcal{T}_{\boldsymbol{n}}$ *are constructed as*

$$\Theta^\uparrow := \big(\underbrace{\theta_1, \dots, \theta_j}_{\textit{scalars}}, \underbrace{x_1 \otimes \mathbf{1}_{k_1}, \dots, x_m \otimes \mathbf{1}_{k_m}}_{\textit{vectors}}, \underbrace{W_1 \otimes \mathbf{1}_{k_{1,\text{out}}} \mathbf{1}_{k_{1,\text{in}}}^\top, \dots, W_\ell \otimes \mathbf{1}_{k_{\ell,\text{out}}} \mathbf{1}_{k_{\ell,\text{in}}}^\top}_{\textit{matrices}}\big).$$

Both the base model and the widened model are trained with the entrywise optimizer using the same base hyperparameter constants $\overline{\gamma}, \overline{\varepsilon}, \overline{\lambda}$*, with per-weight hyperparameters scaled as specified.*

Then, at every training step, both models parametrize the same function. That is, for any training step and any input $\xi \in \mathbb{R}^{d_\text{in}}$*,*

$$f_n(\xi; \Theta) = f_N(\xi; \Theta^\uparrow).$$

C.2.3 Proof of Theorem C.8

We begin by establishing a useful property of the scaled NE⊗OR⊤ program. Consider two runs: one initialized with a set of scalars, vectors, and matrices, and another initialized with the same scalars but with the vectors and matrices widened by duplicating entries. In the second run, the vectors and matrices generated by the program are exactly widened duplications of those obtained in the first run, while the scalars remain unchanged.

Lemma C.9. *Consider a scaled NE⊗OR⊤ program* π *with an initial set* $\boldsymbol{c}^0 \subseteq \boldsymbol{c}$ *of scalars, an initial set* $\boldsymbol{x}^0 \subseteq \boldsymbol{x}$ *of vectors, and an initial set* $\mathcal{W}$ *of matrices. The same scaled NE⊗OR⊤ program can be instantiated on the initial set* $\boldsymbol{c}^0 \subseteq \boldsymbol{c}$ *of scalars, the widened initial set of vectors* ${\boldsymbol{x}^\uparrow}^0 := \{x \otimes \mathbf{1} : x \in \boldsymbol{x}^0\} \subseteq \boldsymbol{x}$*, and the widened initial set of matrices* $\mathcal{W}^\uparrow := \{W \otimes \mathbf{1}_{k_\text{out}} \mathbf{1}_{k_\text{in}}^\top : W \in \mathcal{W}\}$*. Let the two runs of the program generate sequences of vectors and scalars* $(\boldsymbol{x}, \boldsymbol{c})$ *and* $(\boldsymbol{x}^\uparrow, \boldsymbol{c}^\uparrow)$*, respectively. Then* $\boldsymbol{c}^\uparrow = \boldsymbol{c}$*, and* $\boldsymbol{x}^\uparrow = \{x \otimes \mathbf{1} : x \in \boldsymbol{x}\}$*.*

Proof. We show inductively that, in each step of the program, if a scalar is generated, then in both runs, the generated scalars are the same; if a vector is generated, then the vector $x^\uparrow$ generated in the second run and the vector x generated in the first run are related by $x^\uparrow = x \otimes \mathbf{1}$. We consider the cases for different operations of the program.

`Avg`: Let the chosen vector be $x \in \mathbb{R}^n$ in the first run, corresponding to $x^\uparrow = x \otimes \mathbf{1} \in \mathbb{R}^N$ in the second run. Then the `Avg` operation generates the same scalar for the two models

$$\frac{1}{n} \sum_{\alpha=1}^{n} x_\alpha = \frac{1}{N} \sum_{\alpha=1}^{N} x_\alpha^\uparrow \in \mathbb{R}.$$

`MatMul`: Without loss of generality, consider the operation $n_\text{in}^{-1} W x$. The case $n_\text{out}^{-1} W^\top x$ follows in exactly the same manner. Let the chosen matrix and vector be $W \in \mathbb{R}^{n_\text{out} \times n_\text{in}}, x \in \mathbb{R}^{n_\text{in}}$ in the first run, corresponding to $W^{\uparrow} = W \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^{\top} \in \mathbb{R}^{k_{\text{out}}n_{\text{out}}\times k_{\text{in}}n_{\text{in}}}$, $x^{\uparrow} = x \otimes \mathbf{1}_{k_{\text{in}}} \in \mathbb{R}^{k_{\text{in}}n_{\text{in}}}$ in the second run. Then the `MatMul` operation in the first run generates the vector

$$n_{\text{in}}^{-1}Wx,$$

and in the widened model generates the vector

$$(k_{\text{in}}n_{\text{in}})^{-1}(W \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^{\top})(x \otimes \mathbf{1}_{k_{\text{in}}}) = n_{\text{in}}^{-1}Wx \otimes \mathbf{1}_{k_{\text{out}}}.$$

`OuterNonlin`: Suppose the subset of vectors $\tilde{\boldsymbol{x}} \in \mathbb{R}^{n\times|\tilde{\boldsymbol{x}}|}$ in the first run corresponds to $\tilde{\boldsymbol{x}}^{\uparrow} = \tilde{\boldsymbol{x}} \otimes \mathbf{1}_k \in \mathbb{R}^{kn\times|\tilde{\boldsymbol{x}}'|}$ in the second run. Then the `OuterNonlin` operation generates $y^{\uparrow} \in \mathbb{R}^{kn}$ in the second run with

$$\begin{aligned} y^{\uparrow}_{\alpha} &= \frac{1}{(kn)^r}\sum_{\beta_1,\dots,\beta_r=1}^{kn} \psi\left(\tilde{\boldsymbol{x}}^{\uparrow}_{\alpha}; \tilde{\boldsymbol{x}}^{\uparrow}_{\beta_1}; \dots; \tilde{\boldsymbol{x}}^{\uparrow}_{\beta_r}; \boldsymbol{c}\right) \\ &= \frac{1}{n^r}\sum_{\beta_1,\dots,\beta_r=1}^{n} \psi\left(\tilde{\boldsymbol{x}}_{\lceil\alpha/k\rceil}; \tilde{\boldsymbol{x}}_{\beta_1}; \dots; \tilde{\boldsymbol{x}}_{\beta_r}; \boldsymbol{c}\right) = y_{\lceil\alpha/k\rceil}. \end{aligned}$$

where y is the vector generated in the first run. Hence, we have $y^{\uparrow} = y \otimes \mathbf{1}_k$. □

Applying Lemma C.9 to both the forward-pass and the backpropagation program suffices to complete the proof.

Proof of Theorem C.8. We prove the claim by strong induction on training steps t.

At $t = 0$, apply Lemma C.9 to the scaled NE⊗OR⊤ program π that specifies the architecture, instantiated once with the base weights and once with the widened weights. At the final readout (11), the program selects vectors $x^1, \dots, x^{d_{\text{out}}}$ from the set of computed vectors $\boldsymbol{x}$ to form the output. By Lemma C.9, the corresponding vectors in the widened model are $(x^i \otimes \mathbf{1}_{k^{(i)}})_{i=1}^{d_{\text{out}}}$, where $k^{(i)}$ is the width multiplier of $x^i \in \mathbb{R}^{n^{(i)}}$. Since averaging is invariant under duplication, for every $i \in [d_{\text{out}}]$ we have

$$\frac{1}{n^{(i)}}\sum_{\alpha=1}^{n^{(i)}} x^i_{\alpha} \;=\; \frac{1}{n^{(i)}k^{(i)}}\sum_{\alpha=1}^{n^{(i)}k^{(i)}} (x^i \otimes \mathbf{1}_k)_{\alpha},$$

implying $f_n(\xi;\Theta) = f_N(\xi;\Theta^{\uparrow})$ at $t = 0$.

For the induction step, apply Lemma C.9 to the scaled backpropagation programs $dx^1\pi, \dots, dx^{d_{\text{out}}}\pi$. For each $i \in [d_{\text{out}}]$ and for each vector $u \in \mathbb{R}^{n_u}$ in π with widened counterpart $u^{\uparrow} = u \otimes \mathbf{1}_{k_u}$, the generated backpropagated gradients satisfy

$$d^{x^i}u^{\uparrow} \;=\; d^{x^i}u \otimes \mathbf{1}_{k_u} \tag{12}$$

For each scalar c in π, we similarly have

$$d^{x^i}c^{\uparrow} = d^{x^i}c. \tag{13}$$

Let W denote any matrix weight, u any vector weight, and c any scalar weight. Write $y \in \mathbb{R}^{d_{\text{out}}}$ for the model output computed by $y_i = \langle x^i_{\alpha}\rangle_{\alpha}$ according to (11). The gradients used by the optimizer can be written as

$$dW = \frac{\partial \mathcal{L}}{\partial W} = \sum_{j\in d_{\text{out}}}\frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial W}$$

$$= \sum_{j\in d_{\text{out}}} \frac{\partial \mathcal{L}}{\partial y_j} \left(\sum_{z=n_{\text{in}}^{-1}Wh} n_{\text{in}}^{-1} \frac{\partial\langle x_\alpha^j\rangle_\alpha}{\partial z} h^\top + \sum_{z=n_{\text{out}}^{-1}W^\top h} n_{\text{out}}^{-1} \left(\frac{\partial \langle x_\alpha^j\rangle_\alpha}{\partial z} h^\top\right)^\top \right)$$

$$= \sum_{j\in d_{\text{out}}} \frac{\partial \mathcal{L}}{\partial y_j} \left(\sum_{z=n_{\text{in}}^{-1}Wh} n_{\text{in}}^{-1} n_{\text{out}}^{-1} (d^{x^j} z) h^\top + \sum_{z=n_{\text{out}}^{-1}W^\top h} n_{\text{out}}^{-1} n_{\text{in}}^{-1} h (d^{x^j} z)^\top \right)$$

since $d^{x^j} z = n_{\text{out}} \frac{\partial\langle x_\alpha^j\rangle_\alpha}{\partial z_t}$ for $z = n_{\text{in}}^{-1}Wh$. Continuing,

$$du = \frac{\partial \mathcal{L}}{\partial u} = \sum_{j\in d_{\text{out}}} \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial u} = \sum_{j\in d_{\text{out}}} \frac{\partial \mathcal{L}}{\partial y_j} n_u^{-1} d^{x_j} u,$$

$$dc = \frac{\partial \mathcal{L}}{\partial c} = \sum_{j\in d_{\text{out}}} d^{x_j} c.$$

By (12) and (13), the gradients in the widened model and the base model satisfy the relation

$$N_{\text{in}}N_{\text{out}}\, dW^\uparrow \;=\; n_{\text{in}}n_{\text{out}}\, dW \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^\top, \qquad N_u\, du^\uparrow \;=\; n_u\, du \otimes \mathbf{1}_{k_y}, \qquad dc^\uparrow \;=\; dc.$$

Finally, apply the entrywise optimizer to update $(n_{\text{in}}^{-1}W, u, c)$ in the base model and $(N_{\text{in}}^{-1}W^\uparrow, u^\uparrow, c^\uparrow)$ in the widened model, with per-weight hyperparameters scaled as specified by μP. Here we give the proof for vanilla weight decay; the proof for decoupled weight decay is similar, so we do not include it here.

For the matrix weight W, since the gradient of $n_{\text{in}}^{-1}W$ is $n_{\text{in}}dW$, the update rule in the base model is

$$\begin{aligned}&(n_{\text{in}}^{-1}W_{t+1})_{\alpha,\beta}\\ &= (n_{\text{in}}^{-1}W_t)_{\alpha,\beta} - \gamma\, Q_t\Big((n_{\text{in}}dW_0 + \lambda n_{\text{in}}^{-1}W_0)_{\alpha,\beta}, \ldots, (n_{\text{in}}dW_t + \lambda n_{\text{in}}^{-1}W_t)_{\alpha,\beta}; \varepsilon\Big)\end{aligned}$$

Substituting the μP for $\gamma, \lambda, \varepsilon$ and the duplication relations for dW and W, we have by homogeneity of Q_t of degree m

$$\begin{aligned}&= (n_{\text{in}}^{-1}W_t)_{\alpha,\beta} - n_{\text{out}}^{m}n_{\text{in}}^{-1}\overline{\gamma}\; Q_t\Big((n_{\text{in}}dW_0 + n_{\text{in}}n_{\text{out}}^{-1}\overline{\lambda}n_{\text{in}}^{-1}W_0)_{\alpha,\beta}, \ldots,\\ &\qquad\qquad\qquad\qquad (n_{\text{in}}dW_t + n_{\text{in}}n_{\text{out}}^{-1}\overline{\lambda}n_{\text{in}}^{-1}W_t)_{\alpha,\beta};\; n_{\text{out}}^{-1}\overline{\varepsilon}\Big)\\ &= (n_{\text{in}}^{-1}W_t)_{\alpha,\beta} - n_{\text{in}}^{-1}\overline{\gamma}\; Q_t\Big((n_{\text{in}}n_{\text{out}}dW_0 + \overline{\lambda}W_0)_{\alpha,\beta}, \ldots, (n_{\text{in}}n_{\text{out}}dW_t + \overline{\lambda}W_t)_{\alpha,\beta};\; \overline{\varepsilon}\Big)\end{aligned}$$

Equivalently,

$$(W_{t+1})_{\alpha,\beta} = (W_t)_{\alpha,\beta} - \overline{\gamma}Q_t\Big((n_{\text{in}}n_{\text{out}}dW_0 + \overline{\lambda}W_0)_{\alpha,\beta}, \ldots, (n_{\text{in}}n_{\text{out}}dW_t + \overline{\lambda}W_t)_{\alpha,\beta};\; \overline{\varepsilon}\Big).$$

Similarly, the update rule for the matrix weight in the widened model is

$$(W_{t+1}^\uparrow)_{\alpha,\beta} = (W_t^\uparrow)_{\alpha,\beta} - \overline{\gamma}Q_t\Big((N_{\text{in}}N_{\text{out}}dW_0^\uparrow + \overline{\lambda}W_0^\uparrow)_{\alpha,\beta}, \ldots, (N_{\text{in}}N_{\text{out}}dW_t^\uparrow + \overline{\lambda}W_t^\uparrow)_{\alpha,\beta};\; \overline{\varepsilon}\Big).$$

Since for all $s \le t$, by the induction hypothesis we have $N_{\text{in}}N_{\text{out}}dW_s^\uparrow = n_{\text{in}}n_{\text{out}}dW_s \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^\top$ and $W_s^\uparrow = W_s \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^\top$, we conclude that the matrix updates satisfy

$$W_{t+1}^\uparrow = W_{t+1} \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^\top.$$

For the vector weight u, the update rule in the base model is

$$\begin{aligned}(u_{t+1})_\alpha &= (u_t)_\alpha - \gamma\, Q_t\Big((du_0 + \lambda u_0)_\alpha, \dots, (n_u du_t + \lambda u_t)_\alpha; \varepsilon\Big).\\ &= (u_t)_\alpha - n_u^m \overline{\gamma}\, Q_t\Big((du_0 + n_u^{-1}\overline{\lambda} u_0)_\alpha, \dots, (n_u du_t + n_u^{-1}\overline{\lambda} u_t)_\alpha;\ n_u^{-1}\overline{\varepsilon}\Big)\\ &= (u_t)_\alpha - \overline{\gamma}\, Q_t\Big((n_u du_0 + \overline{\lambda} u_0)_\alpha, \dots, (n_u du_t + \overline{\lambda} u_t)_\alpha;\ \overline{\varepsilon}\Big).\end{aligned}$$

Similarly, the update rule in the widened model is

$$(u^{\uparrow}_{t+1})_\alpha = (u^{\uparrow}_t)_\alpha - \overline{\gamma}\, Q_t\Big((N_u du^{\uparrow}_0 + \overline{\lambda} u^{\uparrow}_0)_\alpha, \dots, (N_u du^{\uparrow}_t + \overline{\lambda} u^{\uparrow}_t)_\alpha;\ \overline{\varepsilon}\Big).$$

Since for all $s \le t$, we have $N_u du^{\uparrow}_s = n_u du_s \otimes \mathbf{1}_{k_u}$ and $u^{\uparrow}_s = u_s \otimes \mathbf{1}_{k_u}$, we conclude that the vector updates satisfy

$$u^{\uparrow}_{t+1} = u_{t+1} \otimes \mathbf{1}_{k_u}.$$

Finally, for the scalar weight c, it is immediate that $c^{\uparrow}_{t+1} = c_{t+1}$.

□

C.3 Explanation of μP in Table 2

The μP that we stated in Table 2 is not exactly the same as the versions of μP in Yang and Littwin (2023); Yang et al. (2022). Our convention is intended to facilitate direct comparison with the widening operation that produces the equivalent model in Table 1. Below we compare these conventions with those in the prior papers and justify their equivalence.

One degree of degeneracy. The choice of $A, B, C, D, \tilde{D}, E$ in Table 2 (ignoring the column of output multiplier) exhibits a symmetry with one degree of degeneracy, which we state as a variant of Lemma J.1 in Yang et al. (2022).

Proposition C.10 (Symmetry of scalings)**.** *Consider the setup of an entrywise optimizer with an update function Q_t that is homogeneous of degree m. Suppose we adopt a parametrization prescribed by scalings $A, B, C, D, \tilde{D}, E$ as stated in Table 2.*

For all $\theta > 0$, at any finite width, if we set

$$A \leftarrow A\theta, \quad B \leftarrow B/\theta^2, \quad C \leftarrow C/\theta^{1+m}, \quad D \leftarrow D\theta^2, \quad \tilde{D} \leftarrow \tilde{D}\,\theta^{1+m}, \quad E \leftarrow E\theta,$$

then we obtain a parametrization that is exactly equivalent.

Proof. Let $W = A\theta\,\overline{W}$ with $\overline{W}_{\alpha\beta} \sim \mathcal{N}\left(0, \frac{B}{\theta^2}\,\overline{\sigma}^2\right)$. It follows immediately that $W_{\alpha\beta} \sim \mathcal{N}\left(0, A^2 B\,\overline{\sigma}^2\right)$, which is independent of θ.

The update rule for $\overline{W}$ with learning rate $C\,\theta^{-(1+m)}\overline{\gamma}$, weight decay constant $D\,\theta^2\overline{\lambda}$, decoupled weight decay constant $\tilde{D}\,\theta^{1+m}\overline{\lambda}$, and auxiliary hyperparameter $E\theta\,\overline{\varepsilon}$ is (choose either $D = 0$ or $\tilde{D} = 0$)

$$\begin{aligned}(\overline{W}_{t+1})_{\alpha,\beta} = \big(1 - \overline{\lambda}\,\overline{\gamma}\,C\,\tilde{D}\big)\,(\overline{W}_t)_{\alpha,\beta} - C\,\theta^{-(1+m)}\overline{\gamma}\,Q_t\Big(&\big(d\overline{W}_0 + D\theta^2\overline{\lambda}\,\overline{W}_0\big)_{\alpha,\beta}, \dots,\\ &\big(d\overline{W}_t + D\theta^2\overline{\lambda}\,\overline{W}_t\big)_{\alpha,\beta};\ E\theta\,\overline{\varepsilon}\Big).\end{aligned}$$

Since $W = A\theta\,\overline{W}$, we have $d\overline{W} = A\theta\,dW$, hence

$$(A\theta)^{-1}(W_{t+1})_{\alpha,\beta}$$

$$
\begin{aligned}
&= \big(1 - \overline{\lambda}\,\overline{\gamma}\, C\, \tilde{D}\big)\, (A\theta)^{-1} (W_t)_{\alpha,\beta} \\
&\quad - C\, \theta^{-(1+m)} \overline{\gamma}\; Q_t\Big(\big(A\theta\, dW_0 + D\theta^2 \overline{\lambda}\, (A\theta)^{-1} W_0\big)_{\alpha,\beta}, \ldots, \big(A\theta\, dW_t + D\theta^2 \overline{\lambda}\, (A\theta)^{-1} W_t\big)_{\alpha,\beta};\; E\theta\, \overline{\varepsilon}\Big) \\
&= \big(1 - \overline{\lambda}\,\overline{\gamma}\, C\, \tilde{D}\big)\, (A\theta)^{-1} (W_t)_{\alpha,\beta} \\
&\quad - A^m C \theta^{-1} \overline{\gamma}\; Q_t\Big(\big(dW_0 + \overline{\lambda} D A^{-2} W_0\big)_{\alpha,\beta}, \ldots, \big(dW_t + \overline{\lambda} D A^{-2} W_t\big)_{\alpha,\beta};\; E A^{-1}\, \overline{\varepsilon}\Big)\,, \\
&\hspace{20em} \text{(by the homogeneity of } Q_t \text{ of degree } m\text{)}
\end{aligned}
$$

Multiplying both sides by $A\theta$ yields

$$
\begin{aligned}
(W_{t+1})_{\alpha,\beta} &= \big(1 - \overline{\lambda}\,\overline{\gamma}\, C\, \tilde{D}\big)\, (W_t)_{\alpha,\beta} \\
&\quad - A^{m+1} C\, \overline{\gamma}\; Q_t\Big(\big(dW_0 + \overline{\lambda} D A^{-2} W_0\big)_{\alpha,\beta}, \ldots, \big(dW_t + \overline{\lambda} D A^{-2} W_t\big)_{\alpha,\beta};\; E A^{-1}\, \overline{\varepsilon}\Big)\,,
\end{aligned}
$$

which is independent of θ. This proves that the rescaled parametrization is exactly equivalent. □

Remark C.11. This proposition describes *exact equivalence* at any finite width. We may also consider *asymptotic equivalence*: two scalings $(A, B, C, D, \tilde{D}, E)$ and $(A', B', C', D', \tilde{D}', E')$ are asymptotically equivalent if there exists $\theta > 0$ such that

$$
\frac{A\theta}{A'}, \quad \frac{B/\theta^2}{B'}, \quad \frac{C/\theta^{1+m}}{C'}, \quad \frac{D\theta^2}{D'}, \quad \frac{\tilde{D}\,\theta^{1+m}}{\tilde{D}'}, \quad \frac{E\theta}{E'}
$$

are all $\Theta(1)$ as the width(s) go to ∞ (if there are multiple width dimensions, we assume they go to ∞ at the same rate). The μP scaling is defined via its asymptotic behaviour and hence identifies a family of scalings only up to asymptotic equivalence.

Comparison with μP in Definition 2.9.12 of Yang and Littwin (2023). The *abcd*-parametrization (see Definition 2.9.7 of Yang and Littwin (2023)) considers a general update function Q_t rather than restricting to homogeneous ones. As noted in their Remark 2.2.6, when Q_t is homogeneous of degree m, one should interpret n^{-d} as the scaling for ε and n^{dm-c} as the scaling for the learning rate γ.

In their Definition 2.9.12, where μP is defined, they do not consider weight decay. For vector-like weights, the parametrization translates to

$$
A = 1, \quad B = 1, \quad C = n^m, \quad E = n^{-1}
$$

in our convention. For matrix-like weights, the parametrization translates to

$$
A = 1, \quad B = n^{-1}, \quad C = n^{m-1}, \quad E = n^{-1},
$$

and it does not distinguish n_{out} and n_{in} as we do, because it assumes all widths go to ∞ at the same rate.

Weight decay is treated in Section 2.10.1, where they only consider a decoupled version (note that their notion of "decoupled" differs from ours). There they claim that $\tilde{D}$ should be set so that $C\tilde{D} = 1$. That is, $\tilde{D} = n^{-m}$ for vector-like weights, and $\tilde{D} = n^{1-m}$ for matrix-like weights.

One can verify that this choice of $A, B, C, \tilde{D}, E$ is asymptotically equivalent to what we state in Table 2, except that we add support for vanilla weight decay (controlled by D) and explicitly distinguish n_{in} and n_{out} for matrix-like weights.

Comparison with μP in Table 8 of Yang et al. (2022). The version of μP stated in Yang et al. (2022) does not consider a general entrywise optimizer; instead, it gives parametrizations specifically for SGD and Adam. They do not explicitly discuss weight decay, though it is included in the implementation (we will discuss this later in Appendix C.4). They also do not discuss scaling for ε. So we focus on the comparison of A, B, C.

For vector-like input weights, the first column of Table 8 of Yang et al. (2022) translates to

$$A = 1, \quad B = n_{\text{in}}^{-1}, \quad C = \begin{cases} n_{\text{out}} & \text{for SGD,} \\ 1 & \text{for Adam.} \end{cases}$$

Notice that here n_{in} is constant, while $n_{\text{out}} \to \infty$ (comparable to n in our convention). This is asymptotically equivalent to what we state in the "Vector-like" column of Table 2 (setting $m = 1$ for SGD and $m = 0$ for Adam).

For vector-like output weights, the second column of Table 8 of (Yang et al., 2022) translates to

$$A = 1, \quad B = 1, \quad C = \begin{cases} n_{\text{in}} & \text{for SGD,} \\ 1 & \text{for Adam,} \end{cases}$$

with an additional n_{in}^{-1} output multiplier. In this case $n_{\text{in}} \to \infty$ (comparable to n in our convention). This is exactly equivalent to what we state in the "Vector-like" column of Table 2.

For matrix-like weights, the third column of Table 8 of Yang et al. (2022) translates to

$$A = 1, \quad B = n_{\text{in}}^{-1}, \quad C = \begin{cases} 1 & \text{for SGD,} \\ n_{\text{in}}^{-1} & \text{for Adam.} \end{cases}$$

This is exactly equivalent to what we state in the "Matrix-like" column of Table 2.

C.4 Experimental verification of equivalence

To verify Theorem 2.5, we perform experiments on MLPs, ResNets, and Transformers with variants of SGD, Adam, and AdamW. For each model, we first train a base model from scratch for several epochs under μP, then construct a widened model whose weights are obtained using the rule in Table 1. We next create an optimizer for the widened model and transfer the internal state from the base model to the widened model (explained in Appendix D). Finally, we train both the base and widened models under μP using the same base hyperparameters and compare their equivalence after each training step.

All models pass this verification. We directly use the μP package introduced in Yang et al. (2022), with slight modifications to recover our specific scaling in Table 2 that is essential for exact equivalence. We explain these modifications below.

Modification of the μP package We summarize in Table 3 the exact scaling implemented in the μP package and our minor modifications (highlighted in blue). Specifically, we add support for the default weight decay in Adam, whereas originally only the decoupled weight decay (same as in AdamW) was implemented. Moreover, we implement the correct scaling for ε for Adam and AdamW.

Remark C.12. Below are some further details about this implementation:

1. Table 8 of Yang et al. (2022) claims to document the scalings used in the μP implementation, but it specifies only the entries for A, B, C. The scalings $\tilde{D}$ for Adam and D for SGD were

implemented but not explicitly specified in that paper. We document their explicit forms in the μP implementation and list them in Table 3 (in black and purple). Table 8 of Yang et al. (2022) also states the SGD learning-rate scaling for hidden (matrix) weights to be 1, but the actual implementation uses $n_{\text{out}}/n_{\text{in}}$, which we therefore reflect here.

2. The μP implementation assumes `decoupled_weight_decay=True` in PyTorch for Adam, in which case Adam is equivalent to AdamW. Consequently, it implements only the $\tilde{D}$ scaling and not D. We modify `MuAdam` to also implement the D scaling corresponding to the default, coupled weight decay in Adam.

3. The original implementation does not scale ε, even though later theoretical analysis in Yang and Littwin (2023) recommends scaling ε; see Remark 2.2.6 of Yang and Littwin (2023). We modify `MuAdam` to implement this scaling as in Table 2.

4. The μP implementation requires a specified *base model width*, and all scalings are adjusted so that the base model behaves identically to the standard parametrization. Concretely, this is equivalent to multiplying an additional constant on top of what is shown in Table 3.

	Input vector (inf × fin)	Matrix (inf ×inf)	Output vector (fin ×inf)
Multiplier of weight $W = A \cdot \overline{W}$	1	1	$\frac{1}{n_{\text{in}}}$ (1)
Init variances $\overline{W}_{\alpha\beta} \sim \mathcal{N}\left(0, B \cdot \overline{\sigma}^2\right)$	$\frac{1}{n_{\text{in}}}$	$\frac{1}{n_{\text{in}}}$	1 $(\frac{1}{n_{\text{in}}})$
SGD learning rate scaling $\gamma = C \cdot \overline{\gamma}$	n_{out} (1)	$\frac{n_{\text{out}}}{n_{\text{in}}}$ (1)	n_{in} (1)
SGD weight decay scaling $\lambda = D \cdot \overline{\lambda}$	$\frac{1}{n_{\text{out}}}$ (1)	$\frac{n_{\text{in}}}{n_{\text{out}}}$ (1)	$\frac{1}{n_{\text{in}}}$ (1)
Adam learning rate scaling $\gamma = C \cdot \overline{\gamma}$	1	$\frac{1}{n_{\text{in}}}$ (1)	1
Adam weight decay scaling $\lambda = D \cdot \overline{\lambda}$	$\frac{1}{n_{\text{out}}}$ (1)	$\frac{n_{\text{in}}}{n_{\text{out}}}$ (1)	$\frac{1}{n_{\text{in}}}$ (1)
AdamW weight decay scaling $\lambda = \tilde{D} \cdot \overline{\lambda}$	1	n_{in} (1)	1
Adam, AdamW ε scaling $\varepsilon = E \cdot \overline{\varepsilon}$	$\frac{1}{n_{\text{out}}}$ (1)	$\frac{1}{n_{\text{out}}}$ (1)	$\frac{1}{n_{\text{in}}}$ (1)

Table 3. μP scaling in its implementation. Purple text highlights the key differences between μP scaling and the standard parametrization that is the default in `PyTorch` (written in parentheses in black when different). Blue text highlights our modifications on top of the original implementation, where these scalings were not considered.

D Details of the upscaling algorithm

We explain the omitted details of the upscaling algorithm presented in Meta-Algorithm 1.

Simple example: MLP trained with SGD. For concreteness, we detail the algorithm for the simple example of an MLP trained with vanilla SGD used in Section 2. Note that vanilla SGD does not maintain any internal optimizer state (e.g., momentum), so there is nothing to duplicate or rescale in the optimizer's checkpoint in Step 3 of Meta-Algorithm 1.

Meta-algorithm 2 Upscaling for MLPs under SGD

```
Input: Base MLP weights $(W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}})_{\ell=1}^{L}$ pretrained under $\mu$P; expansion multipliers $(k_\ell \in \mathbb{N})_{\ell=1}^{L-1}$; noise std base constant $\overline{\sigma_\Delta}$; learning-rate base constant $\overline{\gamma^\uparrow}$.
Output: Trained upscaled MLP $({W^\uparrow}^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}})_{\ell=1}^{L}$.
set $k_0 \leftarrow 1,\ k_L \leftarrow 1$; $N_\ell \leftarrow k_\ell n_\ell$ for all $\ell = 0, \dots, L$.
set $\sigma_\Delta^{(1)} \leftarrow \overline{\sigma_\Delta}$, $\sigma_\Delta^{(L)} \leftarrow \overline{\sigma_\Delta}$; $\sigma_\Delta^{(\ell)} \leftarrow N_{\ell-1}^{-1/2}\,\overline{\sigma_\Delta}$ for all $\ell = 2, \dots, L-1$.
  for $\ell = 1$ to $L$ do
    Sample $\Delta^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ with $\Delta_{ij}^{(\ell)} \sim \mathcal{N}\left(0, {\sigma_\Delta^{(\ell)}}^2\right)$.
    ${W^\uparrow}^{(\ell)} \leftarrow k_{\ell-1}^{-1}\, W^{(\ell)} \otimes \mathbf{1}_{k_\ell}\mathbf{1}_{k_{\ell-1}}^\top \; + \; \Delta^{(\ell)}$.
  end for
  Under $\mu$P, run SGD on the upscaled MLP initialized with $({W^\uparrow}^{(\ell)})$, using a $\mu$P-scaled learning rate based on $\overline{\gamma^\uparrow}$.
```

Transfer weights: implementation of Step 1. With the modified μP package in place, we implement weight transfer from the base to the widened model via a generic routine that works for standard architectures that we consider in our experiments. Concretely, the routine operates on both `named_parameters` (model weights) and, when present, `named_buffers` (e.g., BatchNorm running statistics).

For each tensor in `named_parameters` saved in the checkpoint, we query its μP `infshape` to classify it as scalar-like, vector-like, or matrix-like, and then duplicate/rescale according to the rules in the "Widening operation" column of Table 1.

For `named_buffers`, usually simple duplication and copy-over is sufficient. In particular, BatchNorm `running_mean` and `running_var` are vector-like and are duplicated without rescaling, while scalar buffers such as `num_batches_tracked` are copied as-is. Typically no matrix-like buffers occur, so no rescaling is needed.

Transfer optimizer's internal state: implementation of Step 3. One detail not elaborated in Meta-Algorithm 1 is how to transfer the optimizer's internal state, as in Step 3. In Theorem 2.5, for simplicity of analysis, we construct the widened model before any training and show that the widened and base models undergo equivalent updates when the hyperparameters are set appropriately. However, in practical upscaling, we perform widening mid-training. To ensure that the widened model (without adding any noise) is updated equivalently to the base model, as if no upscaling had occurred, it is necessary to transfer the optimizer's internal state.

We implemented a generic routine for transferring optimizer, which works for SGD, Adam, and AdamW (and their variants). Extension to other optimizers is possible but may require additional adjustments. The procedure mirrors the transfer of weights. For each tensor in `named_parameters` saved in the optimizer's checkpoint, we query its μP `infshape` to classify it as scalar-like, vector-like, or matrix-like, and then duplicate and rescale accordingly. Scalar-like tensors are copied as is. For vector-like and matrix-like tensors, in SGD the relevant state is the saved `momentum`, and in Adam/AdamW this is the saved `exp_avg` or `exp_avg_sq`, for the corresponding vector-like or matrix-like weight. Recall that the gradients of the equivalent widened model satisfy

$$dW^\uparrow = k_{\text{out}}^{-1} dW \otimes \mathbf{1}_{k_{\text{out}}}\mathbf{1}_{k_{\text{in}}}^\top \quad \text{if } W \text{ is matrix-like;}$$

$$dW^\uparrow = k^{-1} dW \otimes \mathbf{1}_{k} \quad \text{if } W \text{ is vector-like.}$$

Saved `momentum` and `exp_avg` should be transferred in the same way as the gradient dW. We treat the `exp_avg_sq` case separately, as it requires a distinct rescaling: the scaling factors k_{out}^{-1} or k^{-1} should be squared, consistent with second-moment accumulation.

Practical considerations for weight, buffer, and optimizer-state transfer. As detailed previously, our algorithm requires transferring not only model weights, but also the model's `named_buffers` and the optimizer's internal state. Transferring model weights is the most essential component. Transferring `named_buffers` and optimizer state is needed to guarantee that, with zero added noise, the upscaled model trains exactly as if no upscaling had occurred. In practice, however, especially when upscaling a published model for which optimizer checkpoints may be unavailable, it is acceptable to omit the transfer of `named_buffers` and optimizer state, and performance may remain strong.

E Infinite-width training dynamics for upscaling

We present in Appendix E.1 a modified Ne⊗or⊤ program tailored to analyzing infinite-width limits under mid-training upscaling. In Appendix E.2, we prove its correctness. The proof strategy is to show that the modified program can be expressed as compositions of operations in the original Ne⊗or⊤ program, thereby reducing the analysis to the standard framework and enabling the application of Tensor Program techniques. Finally, to illustrate the usefulness of our modified Ne⊗or⊤ program for analyzing infinite-width limits with upscaling, we work through two simple examples in Appendix E.3 and Appendix E.4, from which we derive several insights.

E.1 Main result: modified Tensor Program

A short recap of the original Ne⊗or⊤ program

The original Ne⊗or⊤ program, introduced in Yang and Littwin (2023), captures infinite-width training dynamics for standard neural network architectures. First, one writes all neural computations during training in the Ne⊗or⊤ language, including both forward and backward passes at each training step. Given this Ne⊗or⊤ program, one then defines the "ket" construction recursively: each vector appearing in the program (for example, layer preactivations) is represented as a random variable (called a "ket") that encodes its distribution as the width $n \to \infty$, and the limiting model outputs or losses at each step are expressed as expectations of functions of these random variables. This reduction yields a deterministic system of "limit equations" describing the evolution of activation and weight distributions over time. Finally, the Master Theorem provides the theoretical foundation, rigorously characterizing the sense in which finite-width dynamics converge to the limiting object defined by the ket construction. We now present our modified Ne⊗or⊤ program, which retains this structure.

Modified Ne⊗or⊤ program language

We characterize the infinite-width limit $n \to \infty$ for the following training-and-upscaling procedure: Consider a network architecture specified in the Ne⊗or⊤ language with hidden width n. After training for T epochs at width n, we upscale to a wider model with hidden width $N = kn$, where k is a fixed constant, and then continue training for an additional T' epochs. (Because we focus on asymptotic training dynamics, it suffices to take all hidden layers to have the same width, as in the original Ne⊗orT construction; this contrasts with Appendix C.2, where we allow heterogeneous hidden widths and expansion factors to study exact finite-width equivalence.)

Rather than extending the language to directly implement the upscaling operation, we seek the minimal modification of Ne⊗orT that preserves its structure as a composition of matrix multiplications and elementwise nonlinearities while enabling the expression of upscaling. By the analysis in Section 2, the training dynamics at width n over the first T epochs can be represented equivalently at width kn. Consequently, instead of changing dimension mid-training, we model the entire procedure at the higher dimension kn by lifting the pre-upscaling stage to that dimension. We therefore define a modified Ne⊗orT program whose only deviation from Definition 2.6.1 of Yang and Littwin (2023) is that all hidden-layer dimensions are nk, where n is the pre-upscaled hidden width and k is the multiplier.

Definition E.1. The modified Ne⊗orT program generates a sequence $\boldsymbol{x}$ of $\mathbb{R}^{nk}$-vectors and a sequence $\boldsymbol{c}$ of $\mathbb{R}$-scalars inductively defined via one of the following ways from an initial set $\boldsymbol{c}^0 \subseteq \boldsymbol{c}$ of random scalars, an initial set $\boldsymbol{x}^0 \subseteq \boldsymbol{x}$ of random $\mathbb{R}^{nk}$ vectors, and an initial set $\mathcal{W}$ of random $\mathbb{R}^{(nk)\times(nk)}$ matrices.

With a slight abuse of notation, we sometimes treat $\boldsymbol{x}, \boldsymbol{c}$ as sets, and at other times regard of $\boldsymbol{c}$ as a vector and $\boldsymbol{x}$ as a matrix with the $\mathbb{R}^{nk}$ vectors as columns; then $\boldsymbol{c}^0$ is just a subvector of $\boldsymbol{c}$ and $\boldsymbol{x}^0$ is a submatrix of $\boldsymbol{x}$. At each step of the program, one can:

`Avg`: Choose a vector $x \in \boldsymbol{x}$ (think of x as a column in $\boldsymbol{x} \in \mathbb{R}^{nk\times|\boldsymbol{x}|}$) and append to $\boldsymbol{c}$ a scalar

$$\langle x_\alpha \rangle_\alpha = \frac{1}{nk}\sum_{\alpha=1}^{nk} x_\alpha \in \mathbb{R}. \tag{14}$$

`MatMul`: Choose a matrix $W \in \mathcal{W}$ and vector $x \in \boldsymbol{x}$, and append to $\boldsymbol{x}$ the vector

$$Wx \in \mathbb{R}^{nk} \quad \text{or} \quad W^\top x \in \mathbb{R}^{nk}.$$

`OuterNonlin`: Choose an integer $r \geq 0$ and a function $\psi : \mathbb{R}^{|\boldsymbol{x}|(r+1)+l} \to \mathbb{R}$, and append to $\boldsymbol{x}$ the vector

$$y \in \mathbb{R}^{nk}, \quad y_\alpha = \langle \psi(\boldsymbol{x}_\alpha; \boldsymbol{x}_{\beta_1}; \ldots; \boldsymbol{x}_{\beta_r}; \boldsymbol{c}) \rangle_{\beta_1,\ldots,\beta_r} = \frac{1}{(nk)^r} \sum_{\beta_1,\ldots,\beta_r=1}^{nk} \psi(\boldsymbol{x}_\alpha; \boldsymbol{x}_{\beta_1}; \ldots; \boldsymbol{x}_{\beta_r}; \boldsymbol{c}) \tag{15}$$

where $\boldsymbol{x}_\gamma \in \mathbb{R}^{|\boldsymbol{x}|}$ is the γ-th row in $\boldsymbol{x}$ as a matrix and $|\boldsymbol{x}|$ is the number of vectors in $\boldsymbol{x}$.

We consider two canonical initialization regimes that encompass both Gaussian and non-Gaussian cases. In contrast to the original setup in Yang and Littwin (2023), we allow two classes of random matrices. The first class consists of matrices of size nk with i.i.d. entries, which model the noise injected during upscaling. The second class consists of matrices of size nk obtained by duplicating a size-n random matrix with i.i.d. entries, which model (an already-widened version of) the weights at the start of training. Analogously, the random vectors at initialization are partitioned into the same two classes. It is straightforward to verify that any network architecture expressible in the original Ne⊗orT program admits an upscaling procedure that can be formulated within the modified Ne⊗orT framework defined here.

Assumption E.2 (Gaussian)**.** The following three hold.

1. The initial set of random matrices is defined as $\mathcal{W} = \mathcal{W}_1 \cup \mathcal{W}_2$. Every entry of each $W \in \mathcal{W}_1$ is sampled i.i.d. from $\mathcal{N}\left(0, \frac{1}{n}\right)$. Each matrix $W' \in \mathcal{W}_2$ is formed as $W' = W \otimes \mathbf{1}_k \mathbf{1}_k^\top$, where W has i.i.d. entries sampled from $\mathcal{N}\left(0, \frac{1}{n}\right)$.

2. The initial set of vectors is defined as $\boldsymbol{x}^0 = \boldsymbol{x}^{0,1} \cup \boldsymbol{x}^{0,2}$. Every entry of each $x \in \boldsymbol{x}^{0,1}$ is sampled i.i.d. from $\mathcal{N}(0,1)$. Each vector $x' \in \boldsymbol{x}^{0,2}$ is formed by $x' = x \otimes \mathbf{1}_k$, where x has i.i.d. entries sampled from $\mathcal{N}(0,1)$.

3. The initial scalars c^0 converge almost surely to 0.

4. All functions ψ used in `OuterNonlin` are pseudo-Lipschitz.

Assumption E.3 (Non-Gaussian)**.** Suppose the same items as Assumption E.2 but with 1) and 4) replaced by the following two.

1*. There exists a sequence $\nu_3, \nu_4, \cdots > 0$ such that (1) each $W \in \mathcal{W}_1$ has independent entries drawn from distributions with zero mean, variance $\frac{1}{n}$ and all higher t-th moment bounded by $\nu_t n^{-t/2}$; (2) each $W' \in \mathcal{W}_2$ is formed by $W' = W \otimes \mathbf{1}_k \mathbf{1}_k^\top$, where W has independent entries drawn from distributions with zero mean, variance $\frac{1}{n}$, and all higher t-th moment bounded by $\nu_t n^{-t/2}$.

4*. All functions ψ used in `OuterNonlin` are polynomially smooth.

We further require initial scalars $\boldsymbol{c}^0$ to have moments of all orders bounded in n.

The "ket" construction

In the infinite-width limit ($n \to \infty$), the training dynamics of a neural network can be analyzed using a calculus of random variables. Without upscaling, in the original NE⊗OR⊤ program, the preactivation vector in each layer is distributed like n i.i.d. random variables. Given a NE⊗OR⊤ program, Yang and Littwin (2023) constructs the associated limit objects using the ket notation, which are random variables that track these preactivations at every step of training. We proceed analogously with upscaling; the difference is that, in this case, each layer's preactivations behave like n blocks consisting of random vectors of dimension k, where these small vectors may have depending entries, but different vectors are i.i.d. Hence, we need to keep track of random k-vectors rather than just scalar random variables.

We first clarify the notation, which is based on but slightly deviates from that in Yang and Littwin (2023).

Notation

- For a vector $x \in \mathbb{R}^{nk}$, for $i \in [k]$ we write

$$x_{(i)} = (x_i, x_{i+k}, x_{i+2k} \dots) \in \mathbb{R}^n, \tag{16}$$

  We are concerned with the situation where x has n i.i.d. k-blocks; equivalently, each $x_{(i)}$ has i.i.d. entries.

  Similarly, for a matrix $W \in \mathbb{R}^{nk \times nk}$, for $i, j \in [k]$ we write

$$W_{(i,j)} := \left(W_{i+(r-1)k,\, j+(c-1)k}\right)_{r,c=1}^{n} \in \mathbb{R}^{n \times n}.$$

  This $W_{(i,j)}$ collects, in an interleaved manner, the (i,j)-th entries from each $k \times k$ block of W across the $n \times n$ grid of blocks. We are concerned with the situation where W has $n \times n$ i.i.d. $k \times k$ blocks; equivalently, each $W_{(i,j)}$ has i.i.d. entries.

- For a vector $x \in \mathbb{R}^{nk}$ with n i.i.d. k-blocks, we write $|x\rangle \in \mathbb{R}^k$ for a random vector such that x look like n i.i.d. samples from $|x\rangle$. We write $\langle x \mid y\rangle$ for the $k \times k$ matrix where

$$\langle x \mid y\rangle_{ij} := \mathbb{E}[|x\rangle_i \, |y\rangle_j] \tag{17}$$

  We are concerned with the situation where x, y look like n i.i.d. samples from $|x\rangle, |y\rangle$ respectively, in the sense that, for any two such vectors $x, y \in \mathbb{R}^{nk}$, we have

$$\lim_{n\to\infty} \frac{(x_{(i)})^\top (y_{(j)})}{n} = \langle x \mid y\rangle_{ij}.$$

  This implies that

$$\lim_{n\to\infty} \frac{x^\top y}{nk} = \frac{\operatorname{Tr}\langle x \mid y\rangle}{k}.$$

  Our notation differs from that of the original Ne⊗or⊤ program: there, $|x\rangle$ denotes a scalar random variable, and $\langle x \mid y\rangle := \mathbb{E}\,|x\rangle\,|y\rangle \in \mathbb{R}$ captures the limit of $x^\top y$. In that setting, one considers the simpler case where $x \in \mathbb{R}^n$ consists of n i.i.d. samples from $|x\rangle$. We further note that our use of $\langle x \mid y\rangle_{ij}$ departs somewhat from the conventional bra–ket notation in physics; we retain this form to remain as close as possible to the original Ne⊗or⊤ program.

  In later parts of the appendix, we will slightly overload the bra–ket notation and use $\langle \cdot \mid \cdot \rangle$ uniformly in both the original and modified Ne⊗or⊤ . The intended meaning should be clear from context—namely, whether $x, y \in \mathbb{R}^n$ or $\mathbb{R}^{nk}$ and whether $|x\rangle, |y\rangle$ are scalars or k-vectors.

- Recall that $\boldsymbol{x}$ represents a matrix with $\mathbb{R}^{nk}$-vectors as columns, collecting all the random vectors that are recursively generated from the modified Ne⊗or⊤ . We write $x^i, i = 1, \dots, |\boldsymbol{x}|$ as its columns (the vectors in the collection), and $\boldsymbol{x}_\alpha, \alpha = 1 \dots, nk$ as its rows (the α-th coordinate of all the vectors in the collection).

- For the current set of vectors $\boldsymbol{x} = (x^1, \dots, x^{|\boldsymbol{x}|}) \in \mathbb{R}^{nk\times|\boldsymbol{x}|}$, we write

$$|\boldsymbol{x}\rangle = \left(\left|x^1\right\rangle, \dots, \left|x^{|\boldsymbol{x}|}\right\rangle\right) \in \mathbb{R}^{k\times|\boldsymbol{x}|}. \tag{18}$$

  We write $|\boldsymbol{x}\rangle_\alpha \in \mathbb{R}^{|\boldsymbol{x}|}$ for the α-th row of $|\boldsymbol{x}\rangle$.

With the notation in place, we now introduce the ket-based construction for our modified Ne⊗or⊤ program. This construction tracks the distribution of preactivations at each layer at every training step in the infinite-width limit.

Definition E.4 (Ket Construction)**.** We recursively define the random k-vector $|x\rangle \in \mathbb{R}^k$ (a multi-vector *ket*) for each vector $x \in \mathbb{R}^{nk}$ and deterministic number $\mathring{\theta}$ for each scalar θ in the program. For a vector Wx produced by `MatMul`, we also define random k-vectors $|Wx\,\hat{\rangle}$ and $|Wx\,\dot{\rangle}$ (called *hat-ket* and *dot-ket* respectively) such that $|Wx\rangle = |Wx\,\hat{\rangle} + |Wx\,\dot{\rangle}$. Their recursive definitions are given below.

- `Init`: For a vector $x \in \boldsymbol{x}^0$, define the ket

$$|x\rangle := \begin{cases} \mathcal{N}(0, I_k) \in \mathbb{R}^k & \text{if } x \in \boldsymbol{x}^{0,1}, \\ \mathcal{N}(0,1) \otimes \mathbf{1}_k \in \mathbb{R}^k & \text{if } x \in \boldsymbol{x}^{0,2}. \end{cases} \tag{19}$$

  Also, for the scalars, let $\mathring{\boldsymbol{c}}^0 := 0 \in \mathbb{R}^{|\boldsymbol{c}^0|}$.

- `Avg`: If θ is generated by `Avg` as in (14), then

$$\mathring{\theta} := \frac{1}{k}\sum_{i=1}^{k} \mathbb{E}\,|x\rangle_i = \frac{1}{k}\operatorname{Tr}\langle x \mid \mathbf{1}\rangle\,, \tag{20}$$

  where the equality follows from our notation of $\langle \bullet \mid \bullet \rangle$ defined in (17).

- `OuterNonlin`: If y is generated by `OuterNonlin` as in (15), then for $\alpha \in [k]$,

$$|y\rangle_\alpha := f(|\boldsymbol{x}\rangle_\alpha) \quad \text{where} \quad f : \mathbb{R}^{|\boldsymbol{x}|} \to \mathbb{R}, \quad f(\boldsymbol{y}) := \frac{1}{k^r}\sum_{\beta_1,\dots,\beta_r=1}^{k} \mathbb{E}\left[\psi(\boldsymbol{y}; |\boldsymbol{x}\rangle^{\boxed{1}}_{\beta_1}; \cdots ; |\boldsymbol{x}\rangle^{\boxed{\mathrm{r}}}_{\beta_r}; \mathring{\boldsymbol{c}})\right]. \tag{21}$$

  Here, as we have mentioned in the notation part, $|\boldsymbol{x}\rangle_\alpha$ refers to the α-th row of the random matrix $|\boldsymbol{x}\rangle \in \mathbb{R}^{k\times|\boldsymbol{x}|}$ that collects the current set of multi-vector kets. Also, $|\boldsymbol{x}\rangle^{\boxed{1}}, \dots, |\boldsymbol{x}\rangle^{\boxed{\mathrm{r}}}$ represents r i.i.d. copies of $|\boldsymbol{x}\rangle$.

- `Hat`: All hat-kets are jointly Gaussian with zero-mean and covariance

$$\operatorname{Cov}\Big(|Wx\,\hat{\rangle}, |Uy\,\hat{\rangle}\Big) = \begin{cases} \mathbb{I}(W = U)\operatorname{Tr}\langle x \mid y\rangle\, I_k & \text{if } W \in \mathcal{W}_1, \\ \mathbb{I}(W = U)(\mathbf{1}_k^\top \langle x \mid y\rangle\, \mathbf{1}_k)\mathbf{1}_k\mathbf{1}_k^\top & \text{if } W \in \mathcal{W}_2. \end{cases} \tag{22}$$

  Here $\mathbb{I}(W = U) = 1$ if and only if W and U are the same matrix as symbol in the program and 0 otherwise.

- `Dot`: By the construction presented here, each entry of any ket, $|x\rangle_\alpha$, for $\alpha \in [k]$, is always a deterministic function of the set of hat-ket entries $|y\,\hat{\rangle}_\beta$, where $\beta \in [k]$. As such, $\frac{\partial |x\rangle_\alpha}{\partial |\bullet\,\hat{\rangle}_\alpha}$ can be defined symbolically. Moreover, it should be independent of α due to permutation symmetry. Hence, we can use $\frac{\partial |x\rangle}{\partial |\bullet\,\hat{\rangle}}$ to denote this value.

  Then, every dot-ket can be expressed as a linear combination of previous kets, represented by the following equation:

$$|Wx\,\dot{\rangle} := \begin{cases} k\sum_{y\in\boldsymbol{x}} |y\rangle\, \mathbb{E}\left[\frac{\partial |x\rangle}{\partial |W^\top y\,\hat{\rangle}}\right] & \text{if } W \in \mathcal{W}_1, \\ k\sum_{y\in\boldsymbol{x}}\sum_{j\in[k]} |y\rangle_j\, \mathbf{1}_k\, \mathbb{E}\left[\frac{\partial |x\rangle}{\partial |W^\top y\,\hat{\rangle}}\right] & \text{if } W \in \mathcal{W}_2. \end{cases} \tag{23}$$

The Master Theorem

Finally, we state the Master theorem for the modified NE⊗OR⊤ , which is practically the same as in Yang and Littwin (2023).

Theorem E.5 (Master theorem)**.** *Consider modified NE⊗OR⊤ with Gaussian or non-Gaussian set-up. Then, as $n \to \infty$, its scalars c satisfy*

$$\boldsymbol{c} \overset{a.s.}{\to} \mathring{\boldsymbol{c}}.$$

In the non-Gaussian set-up, this convergence also happens in L^p for every $p \in [1, \infty)$. In either setup, if the initial scalars are all $\tilde{O}(n^{-1/2})$, then

$$\boldsymbol{c} - \mathring{\boldsymbol{c}} = \tilde{O}(n^{-1/2})$$

as well.

Here, for a random sequence $\boldsymbol{c} = \{\boldsymbol{c}(n)\}_{n \geq 1}$ of fixed-sized vectors, we write $\boldsymbol{c} = \tilde{O}(n^h)$ if $n^{-h-\varepsilon}\boldsymbol{c} \overset{a.s.}{\to} 0$ for every $\varepsilon > 0$.

E.2 Proof of the Master Theorem for the modified Ne⊗or⊤

We now prove the main results stated in the previous section. Our strategy is to construct an original Ne⊗or⊤ program that represents the modified Ne⊗or⊤ program (Definition E.1) under either Assumption E.2 or Assumption E.3. This construction provides an explicit correspondence between the two formulations, enabling a direct transfer of the analytical framework. Consequently, the ket construction for the modified Ne⊗or⊤ (Definition E.4) follows directly from the ket construction of the original Ne⊗or⊤ .

Proof of Theorem E.5. We begin with a program written in the modified Ne⊗or⊤ . Our objective is to iteratively construct an original Ne⊗or⊤ program (Yang and Littwin, 2023) that expresses the same operations and generates equivalent ket construction. We will use $\mathcal{W}, \boldsymbol{x}, \boldsymbol{c}$ to denote the sets of matrices, vectors, and scalars in the modified Ne⊗or⊤ ; and use $\mathcal{W}', \boldsymbol{x}', \boldsymbol{c}'$ to denote the sets in the constructed original Ne⊗or⊤ .

For simplicity, we introduce two additional operations in the original Ne⊗or⊤ : (1) removing a vector from the set $\boldsymbol{x}'$ and (2) removing a scalar from the set $\boldsymbol{c}'$. In the ket construction, the corresponding limiting objects are also removed. These modifications do not affect the expressivity of the original Ne⊗or⊤ ; their primary purpose is to eliminate intermediate quantities and maintain clean sets.

We now consider the cases of different operations of the modified Ne⊗or⊤ to iteratively reconstruct the original Ne⊗or⊤ program. In each step of the construction, we will prove the following property:

P1 The generated scalars and vectors match: $\boldsymbol{c}' = \boldsymbol{c}$, and $\boldsymbol{x}'$ consists of vectors $x_{(i)} \in \mathbb{R}^n$ for $i \in [k], x \in \boldsymbol{x}$. (Recall the notation $x_{(i)}$ defined in (16).) Alternatively as matrices (where we assume appropriate permutation of columns), $\boldsymbol{x}' \in \mathbb{R}^{n \times k|\boldsymbol{x}|}$ is a reshape of $\boldsymbol{x} \in \mathbb{R}^{nk \times |\boldsymbol{x}|}$.

P2 Moreover, the ket constructions match. Firstly, $\mathring{\boldsymbol{c}} = \mathring{\boldsymbol{c}'}$; Secondly, for vector $x \in \boldsymbol{x}$, its ket $|x\rangle \in \mathbb{R}^k$ in the modified Ne⊗or⊤ as defined in Definition E.4 satifies $|x\rangle_i = \left|x_{(i)}\right\rangle$, where $x_{(i)} \in \boldsymbol{x}'$ with $\left|x_{(i)}\right\rangle$ being the corresponding ket in the original Ne⊗or⊤ . Alternatively as matrices (where we assume appropriate permutation of columns), $|\boldsymbol{x}\rangle \in \mathbb{R}^{k \times |\boldsymbol{x}|}$ is a reshape of the vector $|\boldsymbol{x}'\rangle \in \mathbb{R}^{k|\boldsymbol{x}|}$.

The Master Theorem then follows immediately from the Master Theorem of the original Ne⊗or⊤ .

- `Init`: Given initial sets $\boldsymbol{x}^0 \subseteq \mathbb{R}^{nk}$ in the program written in modified Ne⊗or⊤ , we construct an initial set $\boldsymbol{x}^{0'}$ for the original Ne⊗or⊤ in the following way:
  - For each vector $x \in \boldsymbol{x}^{0,1} \subseteq \mathbb{R}^{nk}$, where x has i.i.d. entries drawn from $\mathcal{N}(0, 1)$, we add the vectors $x_{(1)}, \ldots, x_{(k)} \in \mathbb{R}^n$ to the initial set $\boldsymbol{x}^{0'}$.

- For each vector $x \in \boldsymbol{x}^{0,2} \subseteq \mathbb{R}^{nk}$, which consists of constant k-blocks (i.e., $x_{(1)} = \ldots = x_{(k)}$), we add only $x_{(1)}$ to the initial set $\boldsymbol{x}^{0'}$. Later, after the initialization is complete, we will use `OuterNonlin` operations to add $k-1$ identical copies of $x_{(1)}$ to ensure that all of $x_{(1)}, \ldots, x_{(k)}$ are included in $\boldsymbol{x}'$.

Given the initial set of matrices $\mathcal{W}$ in the program written in our Ne⊗or⊤, we construct an initial set $\mathcal{W}'$ for the original Ne⊗or⊤ as follows:

- For each matrix $W \in \mathcal{W}_1 \subseteq \mathbb{R}^{nk \times nk}$, which has i.i.d. entries drawn from $\mathcal{N}(0,1)$, we add the matrices $W_{(1,1)}, \ldots, W_{(k,k)} \in \mathbb{R}^{n \times n}$ to the initial set $\mathcal{W}'$.
- For each matrix $W \in \mathcal{W}_2 \subseteq \mathbb{R}^{nk \times nk}$, which has constant $k \times k$ blocks (i.e., $W_{(1,1)} = \ldots = W_{(k,k)}$), we simply add $W_{(1,1)}$ to the initial set $\mathcal{W}'$.

Finally, for initial scalars $\boldsymbol{c}^0$ in the program written in modified Ne⊗or⊤, we also use it for the initial scalars for the original Ne⊗or⊤. That is, let $\boldsymbol{c}^{0'} := \boldsymbol{c}^0$.

After this step of construction, Property P1 immediately satisfies. For each $x \in \boldsymbol{x}^{0,1}$ in modified Ne⊗or⊤, its ket construction is $|x\rangle \sim \mathcal{N}(0, I_k)$ as defined in (19). This coincides with $(\left|x_{(1)}\right\rangle, \ldots, \left|x_{(k)}\right\rangle)$, the corresponding ket construction in the constructed original Ne⊗or⊤. For each $x \in \boldsymbol{x}^{0,2}$ in modified Ne⊗or⊤, its ket construction is $|x\rangle \sim \mathcal{N}(0,1) \otimes \mathbf{1}_k$ as defined in (19). This again aligns with $(\left|x_{(1)}\right\rangle, \ldots, \left|x_{(k)}\right\rangle)$ from the constructed original Ne⊗or⊤. Hence Property P2 is also satisfied.

- `Avg`: The `Avg` operation in modified Ne⊗or⊤ can be expressed in the original Ne⊗or⊤ as follows: By the inductive hypothesis P1, the chosen vector $x \in \boldsymbol{x}$ has corresponding vectors $x_{(1)}, \ldots, x_{(k)}$ in $\boldsymbol{x}'$. We perform k `Avg` operations on each of these vectors, which appends k scalars to $\boldsymbol{c}'$. Subsequently, we use an additional `OuterNonlin` operation followed by another `Avg` operation to compute the average of the k scalars and append the result to $\boldsymbol{c}'$. Finally, we remove all the intermediate vectors and scalars generated, and leave only the final result in $\boldsymbol{c}'$. This ensure that after this step of construction, Property P1 satisfies.

  The ket construction of the constructed original Ne⊗or⊤ generates deterministic values $\mathbb{E}\left|x_{(1)}\right\rangle, \ldots, \mathbb{E}\left|x_{(k)}\right\rangle$ during the k `Avg` operations. It then adds another deterministic value $\frac{1}{k}\sum_{i=1}^{k} \mathbb{E}\left|x_{(i)}\right\rangle$ from the final `Avg` operation. The removals ensure that only the last scalar is added to $\boldsymbol{c}'$. Meanwhile, the ket construction for `Avg` in the modified Ne⊗or⊤ generates $\frac{1}{k}\sum_{i=1}^{k} \mathbb{E}\,|x\rangle_i$ as defined in (20). By the inductive hypothesis, $|x\rangle_i = \left|x_{(i)}\right\rangle$ for all i. Hence, P2 also satisfies.

- `OuterNonlin`: For each $i \in [k]$, let $\boldsymbol{x}_{(i)}$ denote the set of $\mathbb{R}^n$ vectors formed by taking $x_{(i)}$ for all $x \in \boldsymbol{x}$. Equivalently, consider this as a matrix $\boldsymbol{x}_{(i)} \in \mathbb{R}^{n \times |\boldsymbol{x}|}$ formed by the i-th, $(i+k)$-th, ... rows of $\boldsymbol{x} \in \mathbb{R}^{nk \times |\boldsymbol{x}|}$. By the inducitive hypothesis P1, $\boldsymbol{x}'$ consists of vectors $x_{(i)}, i \in [k], x \in \boldsymbol{x}$. Let $\boldsymbol{x}'_{(i)}$ denote the subset of $\boldsymbol{x}'$ consisting of $\{x_{(i)} : x \in \boldsymbol{x}\}$. As matrices, $\boldsymbol{x}'_{(i)} = \boldsymbol{x}_{(i)}$.

  The `OuterNonlin` operation (15) can be expressed as

$$
\begin{aligned}
y_{(i)} \in \mathbb{R}^n, \quad (y_{(i)})_\alpha &= \frac{1}{(nk)^r} \sum_{i_1,\ldots,i_r=1}^{k} \sum_{\beta_1,\ldots,\beta_r=1}^{n} \psi\left((\boldsymbol{x}_{(i)})_\alpha; (\boldsymbol{x}_{(i_1)})_{\beta_1}; \ldots; (\boldsymbol{x}_{(i_r)})_{\beta_r}; \boldsymbol{c}\right) \\
&= \frac{1}{n^r} \sum_{\beta_1,\ldots,\beta_r=1}^{n} \phi^{(i)}\left(\boldsymbol{x}'_\alpha; \boldsymbol{x}'_{\beta_1}; \ldots; \boldsymbol{x}'_{\beta_r}; \boldsymbol{c}'\right), \quad i = 1, \ldots, k
\end{aligned}
$$

where the function

$$\phi^{(i)} \colon \mathbb{R}^{|\boldsymbol{x}|k(r+1)+l} \to \mathbb{R} \quad \text{is given by}$$

$$\phi^{(i)}(\boldsymbol{y}_0; \ldots; \boldsymbol{y}_r; \boldsymbol{c}) = \frac{1}{k^r} \sum_{i_1,\ldots,i_r=1}^{k} \psi\left(\xi_i(\boldsymbol{y}_0); \xi_{i_1}(\boldsymbol{y}_1); \ldots; \xi_{i_r}(\boldsymbol{y}_r); \boldsymbol{c}'\right).$$

Here $\xi_i : \mathbb{R}^{k|\boldsymbol{x}|} \to \mathbb{R}^{|\boldsymbol{x}|}$ denotes the operation that reshapes the input vector into a matrix in $\mathbb{R}^{k\times|\boldsymbol{x}|}$ and returns its i-th row. This shows that, `OuterNonlin` in our modified NE⊗OR⊤ can be expressed as k `OuterNonlin` operations in the original NE⊗OR⊤ . Moreover, Property P1 is satisfied after this step of construction.

The original NE⊗OR⊤ generates k kets: for $i \in [k]$, $\left|y_{(i)}\right\rangle := f^{(i)}(|\boldsymbol{x}'\rangle)$ where the function $f^{(i)} \colon \mathbb{R}^{k|\boldsymbol{x}|} \to \mathbb{R}$ is given by

$$\begin{aligned} f^{(i)}(\boldsymbol{y}) &= \mathbb{E}\,\phi^{(i)}\left(\boldsymbol{y}; |\boldsymbol{x}'\rangle^{\boxed{1}}; \ldots; |\boldsymbol{x}'\rangle^{\boxed{\text{r}}}; \mathring{\boldsymbol{c}}'\right) \\ &= \frac{1}{k^r} \sum_{i_1,\ldots,i_r=1}^{k} \mathbb{E}\,\psi\left(\xi_i(\boldsymbol{y}); \xi_{i_1}(|\boldsymbol{x}'\rangle^{\boxed{1}}); \ldots; \xi_{i_r}(|\boldsymbol{x}'\rangle^{\boxed{\text{r}}}); \mathring{\boldsymbol{c}}'\right) \end{aligned}$$

By inductive hypothesis P2, we have $\xi_i(|\boldsymbol{x}'\rangle) = \left|\boldsymbol{x}'_{(i)}\right\rangle = |\boldsymbol{x}\rangle_i$, the i-th row of $|\boldsymbol{x}\rangle \in \mathbb{R}^{k\times|\boldsymbol{x}|}$. Moreover, $\mathring{\boldsymbol{c}}' = \mathring{\boldsymbol{c}}$. Hence,

$$= \frac{1}{k^r} \sum_{i_1,\ldots,i_r=1}^{k} \mathbb{E}\,\psi\left(\xi_i(\boldsymbol{y}); |\boldsymbol{x}\rangle_{i_1}^{\boxed{1}}; \ldots; |\boldsymbol{x}\rangle_{i_r}^{\boxed{\text{r}}}; \mathring{\boldsymbol{c}}\right).$$

On the other hand, the modified NE⊗OR⊤ generates ket $|y\rangle$ as defined in (21), with $|y\rangle_i$ exactly matches $\left|y_{(i)}\right\rangle$ from above. Hence, P2 still satisfies after this step of construction.

- `MatMul:` Without loss of generality, consider the matrix multiplication between $W \in \mathcal{W} \subseteq \mathbb{R}^{nk\times nk}$ and $x \in \boldsymbol{x} \subseteq \mathbb{R}^{nk}$. (The case of $W^\top x$ follows in exactly the same way.) Then, for any $i \in [k]$ and $t \in [n]$,

  $$\begin{aligned} \left((Wx)_{(i)}\right)_t = (Wx)_{(t-1)n+i} &= \sum_{s\in[nk]} W_{(t-1)n+i,s} x_s \\ &= \sum_{j\in[k]} \sum_{s\in[n]} (W_{(i,j)})_{t,s} (x_{(j)})_s = \sum_{j\in[k]} (W_{(i,j)} x_{(j)})_t. \end{aligned}$$

  This implies that for all $i \in [k]$,

  $$(Wx)_{(i)} = \sum_{j\in[k]} W_{(i,j)} x_{(j)}.$$

  Case 1: $W \in \mathcal{W}_1$.

  In this case, $W_{(i,j)}$ are i.i.d. matrices for $i, j \in [k]$. `MatMul` in our modified NE⊗OR⊤ can be expressed as k^2 `MatMul` operations in the original NE⊗OR⊤ :

  $$W_{(i,j)} x_{(j)}, \quad i, j \in [k].$$

This is followed by k `OuterNonlin` operations to calculate $\sum_{j\in[k]} W_{(i,j)}x_{(j)}$ for each $i \in [k]$, and to delete the previously appended k^2 vectors $\{W_{(i,j)}x_{(j)} : i, j \in [k]\}$ from $\boldsymbol{x'}$. This ensures that property P1 continues to be satisfied after this step of construction.

The original Ne⊗or⊤ generated the following limiting objects: first, the `MatMul` operations generate k^2 hat-kets, $\left\{\left|W_{(i,j)}x_{(j)}\,\hat{\rangle}\right. : i, j \in [k]\right\}$ and k^2 dot-kets, $\left\{\left|W_{(i,j)}x_{(j)}\,\dot{\rangle}\right. : i, j \in [k]\right\}$. For any Uz in our modified Ne⊗or⊤, and for all $i, j, s, t \in [k]$,

$$\mathrm{Cov}\Big(\left|W_{(i,j)}x_{(j)}\,\hat{\rangle}\right., \left|U_{(s,t)}z_{(t)}\,\hat{\rangle}\right.\Big) = \mathbb{I}(W = U)\mathbf{1}(i = s, j = t)\mathbb{E}\left|x_{(j)}\right\rangle \left|z_{(j)}\right\rangle.$$

Moreover,

$$\begin{aligned}
\left|W_{(i,j)}x_{(j)}\,\dot{\rangle}\right. &:= \sum_{y'\in\boldsymbol{x'}} \left|y'\right\rangle \mathbb{E}\frac{\partial\left|x_{(j)}\right\rangle}{\partial\left|(W_{(i,j)})^\top y'\,\hat{\rangle}\right.}\\
&= \sum_{y'\in\boldsymbol{x'}} \left|y'\right\rangle \mathbb{E}\frac{\partial\left|x_{(j)}\right\rangle}{\partial\left|(W^\top)_{(j,i)} y'\,\hat{\rangle}\right.}\\
&= \sum_{y\in\boldsymbol{x}} \left|y_{(i)}\right\rangle \mathbb{E}\frac{\partial\left|x_{(j)}\right\rangle}{\partial\left|(W^\top)_{(j,i)} y_{(i)}\,\hat{\rangle}\right.}
\end{aligned}$$

since each $y' \in \boldsymbol{x'}$ comes from $y' = y_{(i)}$ for some $y \in \boldsymbol{x}, i \in [k]$, and $(W^\top)_{(j,i)}$ only interacts with $y_{(i)}$.

Following the `MatMul`, the `OuterNonlin` operations generate, for each $i \in [k]$,

$$\left|\textstyle\sum_{j\in[k]} W_{(i,j)}x_{(j)}\right\rangle = \sum_{j\in[k]} \left|W_{(i,j)}x_{(j)}\,\hat{\rangle}\right. + \sum_{j\in[k]} \left|W_{(i,j)}x_{(j)}\,\dot{\rangle}\right..$$

To show Property P2 is still satisfied after this step of construction, we need to check that the kets generated in the modified Ne⊗or⊤, $|Wx\,\hat{\rangle}, |Wx\,\dot{\rangle}$ as defined in (22) and (23) matches the kets generated in the original Ne⊗or⊤:

$$\begin{aligned}
|Wx\,\hat{\rangle} &= \left(\sum_{j\in[k]} \left|W_{(1,j)}x_{(j)}\,\hat{\rangle}\right., \ldots, \sum_{j\in[k]} \left|W_{(k,j)}x_{(j)}\,\hat{\rangle}\right.\right),\\
|Wx\,\dot{\rangle} &= \left(\sum_{j\in[k]} \left|W_{(1,j)}x_{(j)}\,\dot{\rangle}\right., \ldots, \sum_{j\in[k]} \left|W_{(k,j)}x_{(j)}\,\dot{\rangle}\right.\right).
\end{aligned}$$

First, consider the hat part. For any Wz used in the modified Ne⊗or⊤, the corresponding $k \times k$ covariance matrix between the Gaussian vectors on the RHS has the (i, j)-th entry being

$$\mathrm{Cov}\left(\sum_{t\in[k]} \left|W_{(i,t)}x_{(t)}\,\hat{\rangle}\right., \sum_{s\in[k]} \left|W_{(j,s)}z_{(s)}\,\hat{\rangle}\right.\right) = \mathbf{1}(i = j) \sum_{t\in[k]} \mathbb{E}\left|x_{(t)}\right\rangle \left|z_{(t)}\right\rangle = \mathbf{1}(i = j)\,\mathrm{Tr}\,\langle x \mid z\rangle\,.$$

This exactly matches our definition of $\mathrm{Cov}\Big(|Wx\,\hat{\rangle}, |Wz\,\hat{\rangle}\Big)_{ij}$ in (22) for $W \in \mathcal{W}_1$.

Second, consider the dot part. The i-th entry of the vector on the RHS is equal to

$$\sum_{j\in[k]} \left|W_{(i,j)}x_{(j)}\,\dot{\rangle}\right. = \sum_{y\in\boldsymbol{x}} \left|y_{(i)}\right\rangle \sum_{j\in[k]} \mathbb{E}\frac{\partial \left|x_{(j)}\right\rangle}{\partial \left|(W^\top)_{(j,i)}y_{(i)}\,\hat{\rangle}\right.}.$$

By induction hypothesis, $\left|W^\top y\,\hat{\rangle}\right._j = \sum_{i\in[k]} \left|(W^\top)_{(j,i)}y_{(i)}\,\hat{\rangle}\right., |x\rangle_j = \left|x_{(j)}\right\rangle$, and $|y\rangle_i = \left|y_{(i)}\right\rangle$ so

$$\begin{aligned} &= \sum_{y\in\boldsymbol{x}} |y\rangle_i \sum_{j\in[k]} \mathbb{E}\frac{\partial\,|x\rangle_j}{\partial\,|W^\top y\,\hat{\rangle}_j} \\ &= k\sum_{y\in\boldsymbol{x}} |y\rangle_i\, \mathbb{E}\frac{\partial\,|x\rangle_1}{\partial\,|W^\top y\,\hat{\rangle}_1}. \end{aligned}$$

This is exactly the i-th entry of $|Wx\,\dot{\rangle}$ in the modified $\textsc{Ne}\otimes\textsc{or}\top$ by our definition in (23) for $W \in \mathcal{W}_1$.

Case 2: $W \in \mathcal{W}_2$.

In this case, $W_{(i,j)}$ are identical for $i, j \in [k]$. `MatMul` in our modified $\textsc{Ne}\otimes\textsc{or}\top$ can be expressed as, first an `OuterNonlin` operation to calculate, $\sum_{j\in[k]} x_{(j)}$, followed by `MatMul` operations in the original $\textsc{Ne}\otimes\textsc{or}\top$: $W_{(1,1)}\sum_{j\in[k]} x_{(j)}$, and finally delete the previously appended vector. Finally, we use one more `OuterNonlin` operation to add $k-1$ identical copies of the final vector. This ensures that property P1 continues to be satisfied after this step of construction.

The original $\textsc{Ne}\otimes\textsc{or}\top$ generated the following limiting objects: First, the `OuterNonlin` operation generates ket $\left|\sum_{j\in[k]} x_{(j)}\right\rangle = \sum_{j\in[k]} \left|x_{(j)}\right\rangle$; then, `MatMul` operations generate hat-kets, $\left|W_{(1,1)}\sum_j x_{(j)}\,\hat{\rangle}\right.$, and dot-kets, $\left|W_{(1,1)}\sum_j x_{(j)}\,\dot{\rangle}\right.$. For any Uz in our modified $\textsc{Ne}\otimes\textsc{or}\top$, we have

$$\mathrm{Cov}\Big(\left|W_{(1,1)}\textstyle\sum_i x_{(i)}\,\hat{\rangle}\right., \left|U_{(1,1)}\textstyle\sum_j z_{(j)}\,\hat{\rangle}\right.\Big) = \mathbb{I}(W=U)\sum_{i,j}\left\langle x_{(i)} \mid z_{(j)}\right\rangle.$$

Moreover,

$$\begin{aligned} \left|W_{(1,1)}\textstyle\sum_i x_{(i)}\,\dot{\rangle}\right. &:= \sum_{y'\in\boldsymbol{x'}}\sum_{i\in[k]} \left|y'\right\rangle \mathbb{E}\frac{\partial \left|x_{(i)}\right\rangle}{\partial\left|(W_{(1,1)})^\top y'\,\hat{\rangle}\right.} \\ &= \sum_{y\in\boldsymbol{x}}\sum_{j\in[k]}\sum_{i\in[k]} \left|y_{(j)}\right\rangle \mathbb{E}\frac{\partial\left|x_{(i)}\right\rangle}{\partial\left|(W^\top)_{(1,1)}y_{(j)}\,\hat{\rangle}\right.}. \end{aligned}$$

since each $y' \in \boldsymbol{x'}$ comes from $y' = y_{(j)}$ for some $y \in \boldsymbol{x}, j \in [k]$, and $(W^\top)_{(1,1)}$ interacts with every $y_{(j)}$.

Following the `MatMul`, the `OuterNonlin` operations generate a random variable

$$\left|W_{(1,1)}\textstyle\sum_i x_{(i)}\right\rangle = \left|W_{(1,1)}\textstyle\sum_i x_{(i)}\,\hat{\rangle}\right. + \left|W_{(1,1)}\textstyle\sum_i x_{(i)}\,\dot{\rangle}\right..$$

To show Property P2 is still satisfied after this step of construction, we need to check that the kets generated in the modified $\textsc{Ne}\otimes\textsc{or}\top$, $|Wx\,\hat{\rangle}, |Wx\,\dot{\rangle}$ as defined in (22) and (23) matches the kets generated in the original $\textsc{Ne}\otimes\textsc{or}\top$:

$$|Wx\,\hat{\rangle} = \left|W_{(1,1)} \textstyle\sum_i x_{(i)}\,\hat{\rangle}\right.\mathbf{1}_k, \qquad \text{and} \qquad |Wx\,\dot{\rangle} = \left|W_{(1,1)} \textstyle\sum_i x_{(i)}\,\dot{\rangle}\right.\mathbf{1}_k.$$

First, let us consider the hat ket. For any Wz used in the modified $\textsc{Ne}\otimes\textsc{or}\top$, the corresponding $k \times k$ covariance matrix between the Gaussian vectors on the RHS has constant entries which are given by

$$\mathrm{Cov}\left(\left|W_{(1,1)} \textstyle\sum_i x_{(i)}\,\hat{\rangle}\right., \left|W_{(1,1)} \textstyle\sum_j z_{(j)}\,\hat{\rangle}\right.\right) = \sum_{i,j} \left\langle x_{(i)} \mid z_{(j)} \right\rangle = \mathbf{1}_k^\top \langle x \mid z \rangle \, \mathbf{1}_k$$

This exactly matches our definition of $\mathrm{Cov}\left(|Wx\,\hat{\rangle}, |Wz\,\hat{\rangle}\right)$ in (22) for $W \in \mathcal{W}_2$.

Second, let us consider the dot-ket. The vector on the RHS has constant entry, which is

$$\begin{aligned}
\left|W_{(1,1)} \textstyle\sum_i x_{(i)}\,\dot{\rangle}\right. &= \sum_{y \in \boldsymbol{x}} \sum_{j \in [k]} \sum_{i \in [k]} \left|y_{(j)}\right\rangle \mathbb{E} \frac{\partial \left|x_{(i)}\right\rangle}{\partial \left|(W^\top)_{(1,1)} y_{(j)}\,\hat{\rangle}\right.} \\
&= \sum_{y \in \boldsymbol{x}} \sum_{j \in [k]} |y\rangle_j \sum_{i \in [k]} \mathbb{E} \frac{\partial\, |x\rangle_i}{\partial\, |W^\top y\,\hat{\rangle}_i}
\end{aligned}$$

since by induction hypothesis, $|W^\top y\,\hat{\rangle}_j = \left|(W^\top)_{(1,1)} \sum_i y_{(i)}\,\hat{\rangle}\right., |x\rangle_j = \left|x_{(j)}\right\rangle$ and $|y\rangle_j = \left|y_{(j)}\right\rangle$. Continuing,

$$= k \sum_{y \in \boldsymbol{x}} \sum_{j \in [k]} |y\rangle_j \, \mathbb{E} \frac{\partial\, |x\rangle_1}{\partial\, |W^\top y\,\hat{\rangle}_1}.$$

This is exactly the entries of $|Wx\,\dot{\rangle}$ in the modified $\textsc{Ne}\otimes\textsc{or}\top$ (23) for $W \in \mathcal{W}_2$. □

E.3 Example: 3-layer MLP without nonlinearities and with input and output weights frozen

Set-up. Consider a 3-layer MLP without nonlinearity initialized and trained under μP in Table 2. The forward pass is given by

$$h_0^{(1)} = Cx, \quad h^{(2)} = Bh_0^{(1)}, \quad y = \frac{1}{n} Ah^{(2)},$$

where $A \in \mathbb{R}^{1 \times n}, B \in \mathbb{R}^{n \times n}, C \in \mathbb{R}^{n \times 1}$ are the weights. The random initialization at $t = 0$ is

$$(A_0)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma}^2\right), \quad (B_0)_{ij} \sim \mathcal{N}\left(0, n^{-1}\overline{\sigma}^2\right), \quad (C_0)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma}^2\right),$$

We fix the input and output weights A, C and only train the hidden weight B. For simplicitly of notation, we assume full gradient descent on a dataset of size 1, but the same works for any dataset of fixed size; it can also be extended SGD on mini-batches (see Setup 2.3.1 in Yang and Littwin (2023)).

We first train this MLP using vanilla SGD for training steps $t < T$. We write A_t, B_t, C_t, $h_t^{(1)}$, $h_t^{(2)}$, and y_t for the corresponding quantities at step t. We write $A_t^\uparrow, B_t^\uparrow, C_t^\uparrow, h_t^{\uparrow(1)}, h_t^{\uparrow(2)}$ for the equivalent widened version of width $N = nk$, as specified in Table 1. That is,

$$A_t^\uparrow := A_t \otimes \mathbf{1}_k^\top, \quad B_t^\uparrow := k^{-1} B_t \otimes \mathbf{1}_k \mathbf{1}_k^\top, \quad C_t^\uparrow := C_t \otimes \mathbf{1}_k, \quad h_t^{\uparrow(1)} := h_t^{(1)} \otimes \mathbf{1}_k, \quad h_t^{\uparrow(2)} := h_t^{(2)} \otimes \mathbf{1}_k.$$

In the backpropagation, the gradient is computed by

$$dB_t = dh_t^{(2)} (h_t^{(1)})^\top = n^{-1} \mathcal{L}'(y_t) x A_0^\top C_0^\top,$$

The weight is updated by $B_{t+1} = B_t - \overline{\gamma} dB_t$. Other weights A, C are not trained, so we take $A_t = A_0, C_t = C_0$ for all $t < T$.

At step T, we apply upscaling. Specifically, we set

$$A_T^\uparrow := A_{T-1}^\uparrow + \Delta_A, \quad B_T^\uparrow := B_{T-1}^\uparrow + \Delta_B, \quad C_T^\uparrow := C_{T-1}^\uparrow + \Delta_C,$$

where matrices $\Delta_A \in \mathbb{R}^{1\times N}, \Delta_B \in \mathbb{R}^{N \times N}, \Delta_C \in \mathbb{R}^{N \times 1}$ have i.i.d. entries

$$(\Delta_A)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma_{\Delta,A}}^2\right), \quad (\Delta_B)_{ij} \sim \mathcal{N}\left(0, N^{-1}\overline{\sigma_{\Delta,B}}^2\right), \quad (\Delta_C)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma_{\Delta,C}}^2\right).$$

To facilitate theoretical analysis of the role of noise, here we assume that the weights A, B, and C are perturbed by additive noise with potentially different magnitudes.

Finally, we train the upscaled model for training step $t \geq T$. We apply gradient descent on $B_t^\uparrow$.

$$B_{t+1}^\uparrow = B_t^\uparrow - \overline{\gamma^\uparrow} dB_t^\uparrow,$$

where the gradient

$$dB_t^\uparrow = dh_t^{\uparrow(2)} (h_t^{\uparrow(1)})^\top = N^{-1} \mathcal{L}'(y_t) x {A_T^\uparrow}^\top {C_T^\uparrow}^\top.$$

Meanwhile, take $A_t = A_T, C_t = C_T$ for all $t > T$.

Note that the size-n quantities stop to be tracked at step $t = T$. All the size-N quantities are all denoted with $^\uparrow$, and they exist for all training steps.

Before upscaling. We apply the original Ne⊗or⊤ from Yang and Littwin (2023) to analyze the infinite-width training dynamics prior to upscaling. At each training step $t < T$, we have

$$h_t^{(1)} = C_0 x, \quad h_t^{(2)} = \left(B_0 - \overline{\gamma} n^{-1} x \sum_{s=0}^{t-1} \mathcal{L}'(y_s) A_0^\top C_0^\top\right) h_t^{(1)}, \quad y_t = \frac{1}{n} A_0 h_t^{(2)}.$$

The first pre-activation $h_t^{(1)}$ remains constant for all t, and its distribution is tracked by

$$\left|h_t^{(1)}\right\rangle = x \left|C_0\right\rangle \sim \mathcal{N}\left(0, x^2 \overline{\sigma}^2\right).$$

It follows that

$$h_t^{(2)} = B_0 h_0^{(1)} - \overline{\gamma} x^2 \sum_{s=0}^{t-1} \mathcal{L}'(y_s) \frac{C_0^\top C_0}{n} A_0^\top.$$

Hence, the distribution of the second pre-activation $h_t^{(2)}$ is tracked by

$$\left|h_t^{(2)}\right\rangle = \left|B_0 h_0^{(1)}\right\rangle - \overline{\gamma} x^2 \sum_{s=0}^{t-1} \mathcal{L}'(\mathring{y}_s) \underbrace{\langle C_0 \mid C_0 \rangle}_{=\overline{\sigma}^2} \left|A_0\right\rangle,$$

where $\left|B_0 h_0^{(1)}\right\rangle \sim \mathcal{N}\left(0, x^2\overline{\sigma}^4\right)$ and $|A_0\rangle \sim \mathcal{N}\left(0, \overline{\sigma^2}\right)$ are independent. The output y_t converges to the deterministic scalar

$$\mathring{y}_t = \left\langle A_0 \mid h_t^{(2)} \right\rangle = -\overline{\gamma} x^2 \overline{\sigma}^4 \sum_{s=0}^{t-1} \mathcal{L}'(\mathring{y}_s). \tag{24}$$

In this simple model, the infinite-width training dynamics is fully characterized by the recursion (24), and no intermediate distributions need to be tracked.

After upscaling. We analyze the post-upscaling regime using our modified Ne⊗or⊤. At each training step $t \geq T$, we have

$$\begin{aligned}
h_t^{\uparrow(1)} &= (C_0^{\uparrow} + \Delta_C)x, \\
h_t^{\uparrow(2)} &= \Big(B_0^{\uparrow} - \overline{\gamma} N^{-1} x \sum_{s=0}^{T-2} \mathcal{L}'(y_s) A_0^{\uparrow\top} C_0^{\uparrow\top} \\
&\qquad + \Delta_B - \overline{\gamma^{\uparrow}} N^{-1} x \sum_{s=T}^{t-1} \mathcal{L}'(y_s)(A_0^{\uparrow} + \Delta_A)^{\top}(C_0^{\uparrow} + \Delta_C)^{\top}\Big) h_t^{\uparrow(1)}, \\
y_t &= \frac{1}{N}(A_0^{\uparrow} + \Delta_A) h_t^{\uparrow(2)}.
\end{aligned}$$

The first pre-activation $h_t^{\uparrow(1)}$ remains constant for all $t \geq T$, and its distribution is tracked by

$$\left| h_t^{\uparrow(1)} \right\rangle = x \left| C_0^{\uparrow} \right\rangle + x \left| \Delta_C \right\rangle,$$

where

$$\left| C_0^{\uparrow} \right\rangle \sim \mathcal{N}\left(0, \overline{\sigma}^2\right) \mathbf{1}_k, \quad |\Delta_C\rangle \sim \mathcal{N}\left(0, \overline{\sigma_{\Delta,C}}^2 I_k\right).$$

It follows that

$$\begin{aligned}
h_t^{\uparrow(2)} &= B_0^{\uparrow} h_T^{\uparrow(1)} - \overline{\gamma} x \sum_{s=0}^{T-2} \mathcal{L}'(y_s) A_0^{\uparrow\top} \frac{C_0^{\uparrow\top} h_T^{\uparrow(1)}}{N} \\
&\qquad + \Delta_B h_T^{\uparrow(1)} - \overline{\gamma^{\uparrow}} x \sum_{s=T}^{t-1} \mathcal{L}'(y_s)(A_0^{\uparrow} + \Delta_A)^{\top} \frac{(C_0^{\uparrow} + \Delta_C)^{\top} h_T^{\uparrow(1)}}{N}.
\end{aligned}$$

Since $\frac{1}{k}\operatorname{Tr}\left\langle C_0^{\uparrow} \mid h_T^{\uparrow(1)} \right\rangle = x^2\overline{\sigma}^2$ and $\frac{1}{k}\operatorname{Tr}\left\langle C_0^{\uparrow} + \Delta_C \mid h_T^{\uparrow(1)} \right\rangle = x^2(\overline{\sigma}^2 + \overline{\sigma_{\Delta,C}}^2)$, the distribution of the second pre-activation $h_t^{\uparrow(2)}$ is tracked by

$$\begin{aligned}
\left| h_t^{\uparrow(2)} \right\rangle &= \left| B_0^{\uparrow} h_T^{\uparrow(1)} \right\rangle - \overline{\gamma} x^2 \overline{\sigma}^2 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \left| A_0^{\uparrow} \right\rangle \\
&\qquad + \left| \Delta_B h_T^{\uparrow(1)} \right\rangle - \overline{\gamma^{\uparrow}} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,C}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \left(\left| A_0^{\uparrow} \right\rangle + |\Delta_A\rangle \right),
\end{aligned}$$

where $\left|B_0^{\uparrow} h_T^{\uparrow\,(1)}\right\rangle$, $\left|\Delta_B h_T^{\uparrow\,(1)}\right\rangle$, $\left|A_0^{\uparrow}\right\rangle$, and $|\Delta_A\rangle$ are independent Gaussian vectors. The output y_t converges to the deterministic scalar

$$\mathring{y}_t = \left\langle A_0^{\uparrow} \mid h_t^{\uparrow(2)} \right\rangle + \left\langle \Delta_A \mid h_t^{\uparrow(2)} \right\rangle = -\overline{\gamma} x^2 \overline{\sigma}^4 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) - \overline{\gamma^{\uparrow}} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,C}}^2)(\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s). \tag{25}$$

Conclusion. Comparing (24) and (25) shows that the infinite-width training dynamics before and after upscaling coincide if we have the following relation among the hyperparameters:

$$\overline{\gamma}\,\overline{\sigma}^4 = \overline{\gamma^{\uparrow}}(\overline{\sigma}^2 + \overline{\sigma_{\Delta,C}}^2)(\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2).$$

In particular, if $\overline{\sigma_{\Delta,A}} = \overline{\sigma_{\Delta,C}} = 0$ (i.e., no noise added to the frozen weights A or C, with noise added only to B), then for any $\overline{\sigma_{\Delta,B}}$, choosing $\overline{\gamma^{\uparrow}} = \overline{\gamma}$ (the same learning-rate base constant after upscaling) yields equivalent dynamics. Further, even if we add a substantial amount of noise, it is possible to adjust the learning rates accordingly so that the infinite-width limit is preserved. We note that this equivalence does not imply that upscaling is inconsequential, because the above equivalence only pertains to infinite-width limits, and we expect increased width to reduce finite-width errors and thus yield performance closer to those infinite-width limits.

E.4 Example: 4-layer MLP without nonlinearities and with input and output weights frozen

Set-up. We now consider a slightly more complex MLP that adds one additional trainable hidden layer relative to the previous architecture. The forward pass at training step t is

$$h_t^{(1)} = Dx, \quad h_t^{(2)} = Ch_t^{(1)}, \quad h_t^{(3)} = Bh_t^{(2)}, \quad y_t = n^{-1}Ah_t^{(3)},$$

where $A \in \mathbb{R}^{1\times n}$, $B \in \mathbb{R}^{n\times n}$, $C \in \mathbb{R}^{n\times n}$, and $D \in \mathbb{R}^{n\times 1}$ denote the layer weights. At $t = 0$, the random initialization is

$$(A_0)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma}^2\right), \quad (B_0)_{ij} \sim \mathcal{N}\left(0, n^{-1}\overline{\sigma}^2\right), \quad (C_0)_{ij} \sim \mathcal{N}\left(0, n^{-1}\overline{\sigma}^2\right), \quad (D_0)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma}^2\right).$$

We fix the input and output weights A and D and train only the hidden-layer weights B and C.

For steps $t < T$, we optimize the width-n model using vanilla SGD. The gradients are computed via backpropagation as

$$dh_t^{(3)} = n^{-1}\mathcal{L}'(y_t)A_t^{\top}, \quad dh_t^{(2)} = B_t^{\top} dh_t^{(3)}, \quad dh_t^{(1)} = C_t^{\top} dh_t^{(2)},$$

$$dB_t = dh_t^{(3)}(h_t^{(2)})^{\top}, \quad dC_t = dh_t^{(2)}(h_t^{(1)})^{\top},$$

and the weights are updated with learning rate $\overline{\gamma}$ according to

$$B_{t+1} = B_t - \overline{\gamma} dB_t, \quad C_{t+1} = C_t - \overline{\gamma} dC_t.$$

We write $^{\uparrow}$ for the equivalent widened version of width $N = nk$, as specified in Table 1. That is, for $t \leq T$,

$$A_t^{\uparrow} := A_t \otimes \mathbf{1}_k^{\top}, \quad B_t^{\uparrow} := k^{-1}B_t \otimes \mathbf{1}_k\mathbf{1}_k^{\top}, \quad C_t^{\uparrow} := k^{-1}C_t \otimes \mathbf{1}_k\mathbf{1}_k^{\top}, \quad D_t^{\uparrow} := D_t \otimes \mathbf{1}_k;$$

$$h_t^{\uparrow(j)} := h_t^{(j)} \otimes \mathbf{1}_k, \quad dh_t^{\uparrow(j)} := k^{-1} dh_t^{(j)} \otimes \mathbf{1}_k \quad \text{for } j = 1, 2, 3;$$
$$dB_t^{\uparrow} := k^{-1} dB_t \otimes \mathbf{1}_k \mathbf{1}_k^{\top}, \quad dC_t^{\uparrow} := k^{-1} dC_t \otimes \mathbf{1}_k \mathbf{1}_k^{\top}.$$

At step T, we apply upscaling to width N. Specifically, we set

$$A_T^{\uparrow} := A_{T-1}^{\uparrow} + \Delta_A, \quad B_T^{\uparrow} := B_{T-1}^{\uparrow} + \Delta_B, \quad C_T^{\uparrow} := C_{T-1}^{\uparrow} + \Delta_C, \quad D_T^{\uparrow} := D_{T-1}^{\uparrow} + \Delta_D,$$

where the increment matrices $\Delta_A \in \mathbb{R}^{1\times N}$, $\Delta_B \in \mathbb{R}^{N\times N}$, $\Delta_C \in \mathbb{R}^{N\times N}$, and $\Delta_D \in \mathbb{R}^{N\times 1}$ have i.i.d. entries distributed as

$$(\Delta_A)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma_{\Delta,A}}^2\right), \quad (\Delta_B)_{ij} \sim \mathcal{N}\left(0, N^{-1}\overline{\sigma_{\Delta,B}}^2\right),$$
$$(\Delta_C)_{ij} \sim \mathcal{N}\left(0, N^{-1}\overline{\sigma_{\Delta,C}}^2\right), \quad (\Delta_D)_{ij} \sim \mathcal{N}\left(0, \overline{\sigma_{\Delta,D}}^2\right).$$

Finally, for all $t \geq T$, we train the upscaled model. The gradients are computed by

$$dh_t^{\uparrow(3)} = N^{-1}\mathcal{L}'(y_t) A_t^{\uparrow\top}, \quad dh_t^{\uparrow(2)} = B_t^{\uparrow\top} dh_t^{\uparrow(3)}, \quad dh_t^{\uparrow(1)} = C_t^{\uparrow\top} dh_t^{\uparrow(2)},$$
$$dB_t^{\uparrow} = dh_t^{\uparrow(3)} (h_t^{\uparrow(2)})^{\top}, \quad dC_t^{\uparrow} = dh_t^{\uparrow(2)} (h_t^{\uparrow(1)})^{\top},$$

and the weights are updated with learning rate $\overline{\gamma^{\uparrow}}$ as

$$B_{t+1}^{\uparrow} = B_t^{\uparrow} - \overline{\gamma^{\uparrow}} dB_t^{\uparrow}, \quad C_{t+1}^{\uparrow} = C_t^{\uparrow} - \overline{\gamma^{\uparrow}} dC_t^{\uparrow}.$$

Before upscaling. We apply the original Ne⊗or⊤ of Yang and Littwin (2023) to characterize the infinite-width training dynamics prior to upscaling. For each training step $t < T$, the following forward- and backward-propagation relations hold.

$$\begin{aligned}
h_t^{(1)} &= D_0 x, \\
h_t^{(2)} &= \big(C_0 - \overline{\gamma} \sum_{s=0}^{t-1} dh_s^{(2)} (h_s^{(1)})^{\top}\big) h_t^{(1)}, \\
h_t^{(3)} &= \big(B_0 - \overline{\gamma} \sum_{s=0}^{t-1} dh_s^{(3)} (h_s^{(2)})^{\top}\big) h_t^{(2)}, \\
y_t &= n^{-1} A_0 h_t^{(3)}, \\
dh_t^{(3)} &= n^{-1} \mathcal{L}'(y_t) A_t^{\top}, \\
dh_t^{(2)} &= \big(B_0 - \overline{\gamma} \sum_{s=0}^{t-1} dh_s^{(3)} (h_s^{(2)})^{\top}\big)^{\top} dh_t^{(3)}, \\
dh_t^{(1)} &= \big(C_0 - \overline{\gamma} \sum_{s=0}^{t-1} dh_s^{(2)} (h_s^{(1)})^{\top}\big)^{\top} dh_t^{(2)}.
\end{aligned}$$

We track the distributions of the these vectors using kets: [6]

$$\left| h_t^{(1)} \right\rangle = x \left| D_0 \right\rangle \sim \mathcal{N}\left(0, x^2 \overline{\sigma}^2\right),$$

[6] Here $\left| dh_t^{(j)} \right\rangle$ tracks the distribution of $n\, dh_t^{(j)} / \mathcal{L}'(y_t)$ for $j = 1, 2, 3$.

$$\begin{aligned}
\left|h_t^{(2)}\right\rangle &= \left|C_0 h_0^{(1)}\right.\hat{\rangle} - \overline{\gamma}\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\underbrace{\left\langle h_s^{(1)} \mid h_t^{(1)}\right\rangle}_{=x^2\overline{\sigma}^2}\left|dh_s^{(2)}\right\rangle,\\
\left|h_t^{(3)}\right\rangle &= \left|B_0 h_t^{(2)}\right.\hat{\rangle} + \left|B_0 h_t^{(2)}\right.\dot{\rangle} - \overline{\gamma}\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\left\langle h_s^{(2)} \mid h_t^{(2)}\right\rangle\underbrace{\left|dh_s^{(3)}\right\rangle}_{=|A_0\rangle},\\
\mathring{y}_t &= \left\langle A_0 \mid h_t^{(3)}\right\rangle,\\
\left|dh_t^{(3)}\right\rangle &= |A_0\rangle,\\
\left|dh_t^{(2)}\right\rangle &= \left|B_0^\top dh_0^{(3)}\right.\hat{\rangle} - \overline{\gamma}\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\underbrace{\left\langle dh_s^{(3)} \mid dh_t^{(3)}\right\rangle}_{=\overline{\sigma}^2}\left|h_s^{(2)}\right\rangle,\\
\left|dh_t^{(1)}\right\rangle &= \left|C_0^\top dh_t^{(2)}\right.\hat{\rangle} + \left|C_0^\top dh_t^{(2)}\right.\dot{\rangle} - \overline{\gamma}\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\left\langle dh_s^{(2)} \mid dh_t^{(2)}\right\rangle\underbrace{\left|h_s^{(1)}\right\rangle}_{=x|D_0\rangle}.
\end{aligned}$$

By substitution, $\left|h_t^{(2)}\right\rangle$ satisfies the recursion

$$\left|h_t^{(2)}\right\rangle = \left|C_0 h_0^{(1)}\right.\hat{\rangle} - \overline{\gamma}x^2\overline{\sigma}^2\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\left(\left|B_0^\top dh_0^{(3)}\right.\hat{\rangle} - \overline{\gamma}\,\overline{\sigma}^2\sum_{\ell=0}^{s-1}\mathcal{L}'(\mathring{y}_\ell)^2\left|h_\ell^{(2)}\right\rangle\right).$$

Therefore, $\left|h_t^{(2)}\right\rangle$ admits the decomposition

$$\left|h_t^{(2)}\right\rangle = M_t\left|C_0 h_0^{(1)}\right.\hat{\rangle} + N_t\left|B_0^\top dh_0^{(3)}\right.\hat{\rangle}. \tag{26}$$

The coefficients M_t and N_t satisfy the recursion

$$M_t = 1+\overline{\gamma}^2x^2\overline{\sigma}^4\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\sum_{\ell=1}^{s-1}\mathcal{L}'(\mathring{y}_\ell)M_\ell,\quad N_t = -\overline{\gamma}x^2\overline{\sigma}^2\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)+\overline{\gamma}^2x^2\overline{\sigma}^4\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\sum_{\ell=1}^{s-1}\mathcal{L}'(\mathring{y}_\ell)N_\ell. \tag{27}$$

Using the definition of $|\bullet\,\dot{\rangle}$, since $B_0/\overline{\sigma}$ has i.i.d. entries in $\mathcal{N}\left(0, n^{-1}\right)$, we have

$$\left|B_0 h_t^{(2)}\right.\dot{\rangle} = \overline{\sigma}\,\mathbb{E}\left[\frac{\partial\left|h_t^{(2)}\right\rangle}{\partial\left|\overline{\sigma}^{-1}B_0^\top dh_0^{(3)}\right\rangle}\right]\left|dh_0^{(3)}\right\rangle = \overline{\sigma}^2 N_t\,|A_0\rangle\,.$$

Moreover, the bra-ket evaluates to

$$\left\langle h_s^{(2)} \mid h_t^{(2)}\right\rangle = M_s M_t\overline{\sigma}^4 x^2 + N_s N_t\overline{\sigma}^4 =: â‘ .$$

Finally, we obtain

$$\begin{aligned}
\mathring{y}_t &= \overline{\sigma}^2 N_t\,\langle A_0 \mid A_0\rangle - \overline{\gamma}\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)\left\langle h_s^{(2)} \mid h_t^{(2)}\right\rangle\langle A_0 \mid A_0\rangle\\
&= \overline{\sigma}^4 N_t - \overline{\gamma}\,\overline{\sigma}^2\sum_{s=0}^{t-1}\mathcal{L}'(\mathring{y}_s)â‘ .
\end{aligned} \tag{28}$$

After upscaling. We apply the modified Ne⊗or⊤ to characterize the infinite-width training dynamics after upscaling. For $t \geq T$, the forward- and backward-propagation relations are

$$\begin{aligned}
h_t^{\uparrow(1)} &= (D_0^{\uparrow} + \Delta_D)x, \\
h_t^{\uparrow(2)} &= \Big(C_0^{\uparrow} - \overline{\gamma}\sum_{s=0}^{T-2} dh_s^{\uparrow(2)}(h_s^{\uparrow(1)})^{\top} + \Delta_C - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1} dh_s^{\uparrow(2)}(h_s^{\uparrow(1)})^{\top}\Big)h_t^{\uparrow(1)}, \\
h_t^{\uparrow(3)} &= \Big(B_0^{\uparrow} - \overline{\gamma}\sum_{s=0}^{T-2} dh_s^{\uparrow(3)}(h_s^{\uparrow(2)})^{\top} + \Delta_B - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1} dh_s^{\uparrow(3)}(h_s^{\uparrow(2)})^{\top}\Big)h_t^{\uparrow(2)}, \\
y_t &= N^{-1}(A_0^{\uparrow} + \Delta_A)h_t^{\uparrow(3)}, \\
dh_t^{\uparrow(3)} &= N^{-1}\mathcal{L}'(y_t)(A_0^{\uparrow} + \Delta_A)^{\top}, \\
dh_t^{\uparrow(2)} &= \Big(B_0^{\uparrow} - \overline{\gamma}\sum_{s=0}^{T-2} dh_s^{\uparrow(3)}(h_s^{\uparrow(2)})^{\top} + \Delta_B - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1} dh_s^{\uparrow(3)}(h_s^{\uparrow(2)})^{\top}\Big)^{\top} dh_T^{\uparrow\,(3)}, \\
dh_t^{\uparrow(1)} &= \Big(C_0^{\uparrow} - \overline{\gamma}\sum_{s=0}^{T-2} dh_s^{\uparrow(2)}(h_s^{\uparrow(1)})^{\top} + \Delta_C - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1} dh_s^{\uparrow(2)}(h_s^{\uparrow(1)})^{\top}\Big)^{\top} dh_t^{\uparrow(2)}.
\end{aligned}$$

We track the distributions of the preactivations and backpropagated signals using the multi-vector kets: [7]

$$\begin{aligned}
\left|h_t^{\uparrow(1)}\right\rangle &= x\left(\left|D_0^{\uparrow}\right\rangle + \left|\Delta_D\right\rangle\right), \\
\left|h_t^{\uparrow(2)}\right\rangle &= \left|C_0^{\uparrow}h_T^{\uparrow\,(1)}\right.\hat{\Big\rangle} - \overline{\gamma}\sum_{s=0}^{T-2}\mathcal{L}'(\mathring{y}_s)\underbrace{\frac{\operatorname{Tr}\left\langle h_s^{\uparrow(1)} \mid h_T^{\uparrow\,(1)}\right\rangle}{k}}_{=x^2\overline{\sigma}^2}\left|dh_s^{\uparrow(2)}\right\rangle \\
&\quad + \left|\Delta_C h_T^{\uparrow\,(1)}\right.\hat{\Big\rangle} - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1}\mathcal{L}'(\mathring{y}_s)\underbrace{\frac{\operatorname{Tr}\left\langle h_s^{\uparrow(1)} \mid h_T^{\uparrow\,(1)}\right\rangle}{k}}_{=x^2(\overline{\sigma}^2+\overline{\sigma_{\Delta,D}}^2)}\left|dh_s^{\uparrow(2)}\right\rangle, \\
\left|h_t^{\uparrow(3)}\right\rangle &= \left|B_0^{\uparrow}h_t^{\uparrow(2)}\right.\hat{\Big\rangle} + \left|B_0^{\uparrow}h_t^{\uparrow(2)}\right.\dot{\Big\rangle} - \overline{\gamma}\sum_{s=0}^{T-2}\mathcal{L}'(\mathring{y}_s)\frac{\operatorname{Tr}\left\langle h_s^{\uparrow(2)} \mid h_t^{\uparrow(2)}\right\rangle}{k}\underbrace{\left|dh_s^{\uparrow(3)}\right\rangle}_{=\left|A_0^{\uparrow}\right\rangle} \\
&\quad + \left|\Delta_B h_t^{\uparrow(2)}\right.\hat{\Big\rangle} + \left|\Delta_B h_t^{\uparrow(2)}\right.\dot{\Big\rangle} - \overline{\gamma^{\uparrow}}\sum_{s=T}^{t-1}\mathcal{L}'(\mathring{y}_s)\frac{\operatorname{Tr}\left\langle h_s^{\uparrow(2)} \mid h_t^{\uparrow(2)}\right\rangle}{k}\underbrace{\left|dh_s^{\uparrow(3)}\right\rangle}_{=\left|A_0^{\uparrow}\right\rangle+\left|\Delta_A\right\rangle}, \\
\mathring{y}_t &= \frac{\operatorname{Tr}\left\langle A_0^{\uparrow} \mid h_t^{\uparrow(3)}\right\rangle + \operatorname{Tr}\left\langle \Delta_A \mid h_t^{\uparrow(3)}\right\rangle}{k},
\end{aligned}$$

[7] As before, $\left|dh_t^{\uparrow(j)}\right\rangle$ tracks the distribution of $N\,dh_t^{\uparrow(j)}/\mathcal{L}'(y_t)$ for $j = 1, 2, 3$.

$$\left|dh_t^{\uparrow(3)}\right\rangle = \left|A_0^{\uparrow}\right\rangle + \left|\Delta_A\right\rangle,$$

$$\left|dh_t^{\uparrow(2)}\right\rangle = \left|B_0^{\uparrow\top} dh_T^{\uparrow(3)}\right.\hat{\Big\rangle} - \overline{\gamma} \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \underbrace{\frac{\operatorname{Tr}\left\langle dh_s^{\uparrow(3)} \mid dh_t^{\uparrow(3)}\right\rangle}{k}}_{=\overline{\sigma}^2} \left|h_s^{\uparrow(2)}\right\rangle$$

$$+ \left|\Delta_B^{\top} dh_T^{\uparrow(3)}\right\rangle - \overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \underbrace{\frac{\operatorname{Tr}\left\langle dh_s^{\uparrow(3)} \mid dh_t^{\uparrow(3)}\right\rangle}{k}}_{=\overline{\sigma}^2+\overline{\sigma_{\Delta,A}}^2} \left|h_s^{\uparrow(2)}\right\rangle,$$

$$\left|dh_t^{\uparrow(1)}\right\rangle = \left|C_0^{\uparrow\top} dh_t^{\uparrow(2)}\right.\hat{\Big\rangle} + \left|C_0^{\uparrow\top} dh_t^{\uparrow(2)}\right.\dot{\Big\rangle} - \overline{\gamma} \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \frac{\operatorname{Tr}\left\langle dh_s^{\uparrow(2)} \mid dh_t^{\uparrow(2)}\right\rangle}{k} \underbrace{\left|h_s^{\uparrow(1)}\right\rangle}_{=x\left|D_0^{\uparrow}\right\rangle}$$

$$+ \left|\Delta_C^{\top} dh_t^{\uparrow(2)}\right.\hat{\Big\rangle} + \left|\Delta_C^{\top} dh_t^{\uparrow(2)}\right.\dot{\Big\rangle} - \overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \frac{\operatorname{Tr}\left\langle dh_s^{\uparrow(2)} \mid dh_t^{\uparrow(2)}\right\rangle}{k} \underbrace{\left|h_s^{\uparrow(1)}\right\rangle}_{=x(\left|D_0^{\uparrow}\right\rangle+|\Delta_D\rangle)}.$$

By substitution, $\left|h_t^{\uparrow(2)}\right\rangle$ satisfies the recursion

$$\begin{aligned}\left|h_t^{\uparrow(2)}\right\rangle = {} & \left|C_0^{\uparrow} h_T^{\uparrow(1)}\right.\hat{\Big\rangle} - \overline{\gamma} x^2 \overline{\sigma}^2 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \left(\left|B_0^{\uparrow\top} dh_0^{\uparrow(3)}\right.\hat{\Big\rangle} - \overline{\gamma}\, \overline{\sigma}^2 \sum_{\ell=0}^{s-1} \mathcal{L}'(\mathring{y}_\ell) \left|h_\ell^{\uparrow(2)}\right\rangle \right) + \left|\Delta_C h_T^{\uparrow(1)}\right.\hat{\Big\rangle} \\ & - \overline{\gamma^{\uparrow}} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \Bigg(\left|B_0^{\uparrow\top} dh_T^{\uparrow(3)}\right.\hat{\Big\rangle} - \overline{\gamma}\, \overline{\sigma}^2 \sum_{\ell=0}^{T-2} \mathcal{L}'(\mathring{y}_\ell) \left|h_\ell^{\uparrow(2)}\right\rangle + \left|\Delta_B^{\top} dh_T^{\uparrow(3)}\right.\hat{\Big\rangle} \\ & \qquad - \overline{\gamma^{\uparrow}} (\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{\ell=T}^{s-1} \mathcal{L}'(\mathring{y}_\ell) \left|h_\ell^{\uparrow(2)}\right\rangle \Bigg).\end{aligned}$$

Recall from (26) that for $t \leq T-2$ we have

$$\left|h_t^{(2)}\right\rangle = M_t \left|C_0 h_0^{(1)}\right.\hat{\Big\rangle} + N_t \left|B_0^{\top} dh_0^{(3)}\right.\hat{\Big\rangle},$$

which translates to

$$\left|h_t^{\uparrow(2)}\right\rangle = M_t \left|C_0^{\uparrow} h_0^{\uparrow(1)}\right.\hat{\Big\rangle} + N_t \left|B_0^{\uparrow\top} dh_0^{\uparrow(3)}\right.\hat{\Big\rangle}.$$

Therefore, for $t \geq T$, we obtain the decomposition

$$\begin{aligned}\left|h_t^{\uparrow(2)}\right\rangle = {} & M_t \left|C_0^{\uparrow} h_0^{\uparrow(1)}\right.\hat{\Big\rangle} + M'_t \left|C_0^{\uparrow} h_T^{\uparrow(1)}\right.\hat{\Big\rangle} + M'_t \left|\Delta_C h_T^{\uparrow(1)}\right.\hat{\Big\rangle} \\ & + N_t \left|B_0^{\uparrow\top} dh_0^{\uparrow(3)}\right.\hat{\Big\rangle} + N'_t \left|B_0^{\uparrow\top} dh_T^{\uparrow(3)}\right.\hat{\Big\rangle} + N'_t \left|\Delta_B^{\top} dh_T^{\uparrow(3)}\right.\hat{\Big\rangle}.\end{aligned}$$

The coefficients satisfy

$$\begin{aligned}
M_t &= \overline{\gamma}^2 x^2 \overline{\sigma}^4 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \sum_{\ell=1}^{s-1} \mathcal{L}'(\mathring{y}_\ell) M_\ell \\
&\quad + \overline{\gamma^\uparrow} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \left(\overline{\gamma}\, \overline{\sigma}^2 \sum_{\ell=0}^{T-2} \mathcal{L}'(\mathring{y}_\ell) M_\ell + \overline{\gamma^\uparrow} (\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{\ell=T}^{s-1} \mathcal{L}'(\mathring{y}_\ell) M_\ell \right), \\
M'_t &= 1 + \overline{\gamma^\uparrow}^2 x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2)(\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \sum_{\ell=T}^{s-1} \mathcal{L}'(\mathring{y}_\ell) M'_\ell, \\
N_t &= -\overline{\gamma} x^2 \overline{\sigma}^2 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) + \overline{\gamma}^2 x^2 \overline{\sigma}^4 \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s) \sum_{\ell=1}^{s-1} \mathcal{L}'(\mathring{y}_\ell) N_\ell \\
&\quad + \overline{\gamma^\uparrow} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \left(\overline{\gamma} \overline{\sigma}^2 \sum_{\ell=0}^{T-2} \mathcal{L}'(\mathring{y}_\ell) N_\ell + \overline{\gamma^\uparrow} (\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{\ell=T}^{s-1} \mathcal{L}'(\mathring{y}_\ell) N_\ell \right), \\
N'_t &= -\overline{\gamma^\uparrow} x^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2) \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s) \left(1 - \overline{\gamma^\uparrow} (\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \sum_{\ell=T}^{s-1} \mathcal{L}'(\mathring{y}_\ell) N'_\ell \right).
\end{aligned} \tag{29}$$

Using the definition of $|\bullet\,\dot{\rangle}$ in (23), and since $kB_0^\uparrow/\overline{\sigma} \in \mathcal{W}_2$ and $\sqrt{k}\,\Delta_B/\overline{\sigma_{\Delta,B}} \in \mathcal{W}_1$, we have

$$\begin{aligned}
\left| B_0^\uparrow h_t^{\uparrow(2)} \dot{\right\rangle} &= k^{-1}\overline{\sigma} k \ \mathbb{E}\left[\frac{\partial \left| h_t^{\uparrow(2)} \right\rangle}{\partial \left| k\overline{\sigma}^{-1} {B_0^\uparrow}^\top dh_0^{\uparrow(3)} \hat{\right\rangle}} \right] \sum_j \left| dh_0^{\uparrow(3)} \right\rangle_j \mathbf{1}_k \\
&\quad + k^{-1}\overline{\sigma} k \ \mathbb{E}\left[\frac{\partial \left| h_t^{\uparrow(2)} \right\rangle}{\partial \left| k\overline{\sigma}^{-1} {B_0^\uparrow}^\top d{h_T^\uparrow}^{(3)} \hat{\right\rangle}} \right] \sum_j \left| d{h_T^\uparrow}^{(3)} \right\rangle_j \mathbf{1}_k \\
&= \overline{\sigma} \left(k^{-1}\overline{\sigma} N_t \sum_j \left| dh_0^{\uparrow(3)} \right\rangle_j \mathbf{1}_k + k^{-1}\overline{\sigma} N'_t \sum_j \left| d{h_T^\uparrow}^{(3)} \right\rangle_j \mathbf{1}_k \right) \\
&= \overline{\sigma}^2 N_t \left| A_0^\uparrow \right\rangle + \overline{\sigma}^2 N'_t \left(\left| A_0^\uparrow \right\rangle + \frac{\sum_j |\Delta_A\rangle_j}{k} \mathbf{1}_k \right), \\
\left| \Delta_B h_t^{\uparrow(2)} \dot{\right\rangle} &= \overline{\sigma_{\Delta,B}} k^{-1/2} k \ \mathbb{E}\left[\frac{\partial \left| h_t^\uparrow \right\rangle^{(2)}}{\partial \left| k^{1/2} \overline{\sigma_{\Delta,B}}^{-1} \Delta_B^\top d{h_T^\uparrow}^{(3)} \right\rangle} \right] \left| d{h_T^\uparrow}^{(3)} \right\rangle \\
&= \overline{\sigma_{\Delta,B}} k^{-1/2} k N'_t k^{-1/2} \overline{\sigma_{\Delta,B}} \left(\left| A_0^\uparrow \right\rangle + |\Delta_A\rangle \right) \\
&= \overline{\sigma_{\Delta,B}}^2 N'_t \left(\left| A_0^\uparrow \right\rangle + |\Delta_A\rangle \right).
\end{aligned}$$

Moreover, for $s \leq T-2$ and $t \geq T$, we have

$$\begin{aligned}
&\left\langle h_s^{\uparrow(2)} \mid h_t^{\uparrow(2)} \right\rangle \\
&= M_s M_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle h_0^{\uparrow(1)} \mid h_0^{\uparrow(1)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top + M_s M'_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle h_0^{\uparrow(1)} \mid h_T^{\uparrow\,(1)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&\quad + N_s N_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle dh_0^{\uparrow(3)} \mid dh_0^{\uparrow(3)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top + N_s N'_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle dh_0^{\uparrow(3)} \mid dh_T^{\uparrow\,(3)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&= \left(M_s (M_t + M'_t) \overline{\sigma}^4 x^2 + N_s (N_t + N'_t) \overline{\sigma}^4 \right) \mathbf{1}_k \mathbf{1}_k^\top .
\end{aligned}$$

Hence,

$$\frac{\operatorname{Tr} \left\langle h_s^{\uparrow(2)} \mid h_t^{\uparrow(2)} \right\rangle}{k} = M_s (M_t + M'_t) \overline{\sigma}^4 x^2 + N_s (N_t + N'_t) \overline{\sigma}^4 =: \textcircled{1'}.$$

Similarly, for $s, t \geq T$, we have

$$\begin{aligned}
&\left\langle h_s^{\uparrow(2)} \mid h_t^{\uparrow(2)} \right\rangle \\
&= M_s M_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle h_0^{\uparrow(1)} \mid h_0^{\uparrow(1)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top + N_s N_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle dh_0^{\uparrow(3)} \mid dh_0^{\uparrow(3)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&\quad + M'_s M'_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle h_T^{\uparrow\,(1)} \mid h_T^{\uparrow\,(1)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top + N'_s N'_t \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle dh_T^{\uparrow\,(3)} \mid dh_T^{\uparrow\,(3)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&\quad + M'_s M'_t \overline{\sigma_{\Delta,C}}^2 k^{-1} \operatorname{Tr} \left\langle h_T^{\uparrow\,(1)} \mid h_T^{\uparrow\,(1)} \right\rangle I_k + N'_s N'_t \overline{\sigma_{\Delta,B}}^2 k^{-1} \operatorname{Tr} \left\langle dh_T^{\uparrow\,(3)} \mid dh_T^{\uparrow\,(3)} \right\rangle I_k \\
&\quad + (M_s M'_t + M'_s M_t) \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle h_0^{\uparrow(1)} \mid h_T^{\uparrow\,(1)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&\quad + (N_s N'_t + N'_s N_t) \overline{\sigma}^2 k^{-2} \left(\mathbf{1}_k^\top \left\langle dh_0^{\uparrow(3)} \mid dh_T^{\uparrow\,(3)} \right\rangle \mathbf{1}_k \right) \mathbf{1}_k \mathbf{1}_k^\top \\
&= \big((M_s M_t + M_s M'_t + M'_s M_t) \overline{\sigma}^4 x^2 + (N_s N_t + N_s N'_t + N'_s N_t) \overline{\sigma}^4 \\
&\quad + M'_s M'_t \overline{\sigma}^2 (\overline{\sigma}^2 + k^{-1} \overline{\sigma_{\Delta,D}}^2) x^2 + N'_s N'_t \overline{\sigma}^2 (\overline{\sigma}^2 + k^{-1} \overline{\sigma_{\Delta,A}}^2) \big) \mathbf{1}_k \mathbf{1}_k^\top \\
&\quad + \left(M'_s M'_t \overline{\sigma_{\Delta,C}}^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2) x^2 + N'_s N'_t \overline{\sigma_{\Delta,B}}^2 (\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2) \right) I_k .
\end{aligned}$$

Consequently,

$$\begin{aligned}
\frac{\operatorname{Tr} \left\langle h_s^{(2)} \mid h_t^{(2)} \right\rangle}{k} &= x^2 \overline{\sigma}^4 (M_s + M'_s)(M_t + M'_t) + \overline{\sigma}^4 (N_s + N'_s)(N_t + N'_t) \\
&\quad + M'_s M'_t \left(k^{-1} \overline{\sigma}^2 \overline{\sigma_{\Delta,D}}^2 + \overline{\sigma}^2 \overline{\sigma_{\Delta,C}}^2 + \overline{\sigma_{\Delta,C}}^2 \overline{\sigma_{\Delta,D}}^2 \right) x^2 \\
&\quad + N'_s N'_t \left(k^{-1} \overline{\sigma}^2 \overline{\sigma_{\Delta,A}}^2 + \overline{\sigma}^2 \overline{\sigma_{\Delta,B}}^2 + \overline{\sigma_{\Delta,A}}^2 \overline{\sigma_{\Delta,B}}^2 \right) \\
&=: \textcircled{2'}.
\end{aligned}$$

Finally, by independence of various hat-kets, we obtain the post-upscaling readout

$$
\begin{aligned}
\mathring{y}_t &= \frac{\operatorname{Tr}\left\langle A_0^{\uparrow} \mid h_t^{\uparrow(3)} \right\rangle + \operatorname{Tr}\left\langle \Delta_A \mid h_t^{\uparrow(3)} \right\rangle}{k} \\
&= k^{-1} \operatorname{Tr}\left(\left(-\overline{\gamma} \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s)\textcircled{1'} - \overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s)\textcircled{2'} + \overline{\sigma}^2 (N_t + N_t') + \overline{\sigma_{\Delta,B}}^2 N_t' \right) \left\langle A_0^{\uparrow} \mid A_0^{\uparrow} \right\rangle \right) \\
&\quad + k^{-1} \operatorname{Tr}\left(\left(-\overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s)\textcircled{2'} + \overline{\sigma_{\Delta,B}}^2 N_t' \right) \langle \Delta_A \mid \Delta_A \rangle + \overline{\sigma}^2 N_t' \left\langle k^{-1} \sum_j (\Delta_A)_j \mathbf{1}_k \mid \Delta_A \right\rangle \right) \\
&= \overline{\sigma}^2 \left(-\overline{\gamma} \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s)\textcircled{1'} - \overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s)\textcircled{2'} + \overline{\sigma}^2 (N_t + N_t') + \overline{\sigma_{\Delta,B}}^2 N_t' \right) \\
&\quad + \overline{\sigma_{\Delta,A}}^2 \left(-\overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s)\textcircled{2'} + N_t' (k^{-1} \overline{\sigma}^2 + \overline{\sigma_{\Delta,B}}^2) \right) \\
&= \overline{\sigma}^4 \left(N_t + N_t' \right) + (\overline{\sigma}^2 \overline{\sigma_{\Delta,B}}^2 + k^{-1} \overline{\sigma}^2 \overline{\sigma_{\Delta,A}}^2 + \overline{\sigma_{\Delta,A}}^2 \overline{\sigma_{\Delta,B}}^2) N_t' \\
&\quad - \overline{\sigma}^2 \overline{\gamma} \sum_{s=0}^{T-2} \mathcal{L}'(\mathring{y}_s)\textcircled{1'} - \left(\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2 \right) \overline{\gamma^{\uparrow}} \sum_{s=T}^{t-1} \mathcal{L}'(\mathring{y}_s)\textcircled{2'}.
\end{aligned}
\tag{30}
$$

Conclusion. Adding one more hidden layer immediately complicates the computation of the infinite-width dynamics, but the system remains analyzable. Since in the previous section we found for a simpler model that, even when adding noise while upscaling, it is possible to exactly maintain the infinite-width limit of training dynamics, it is natural to ask whether that remains possible in this more complicated architecture.

Comparing the coefficient recursions before and after upscaling, (27) and (29), we observe that if

$$\overline{\gamma^{\uparrow}}^2 \left(\overline{\sigma}^2 + \overline{\sigma_{\Delta,D}}^2 \right) \left(\overline{\sigma}^2 + \overline{\sigma_{\Delta,A}}^2 \right) = \overline{\gamma}^2 \, \overline{\sigma}^4,$$

an analogous relation to that from the previous section, then the recursions for $(M_t + M_t')$ and $(N_t + N_t')$ coincide with their pre-upscaling counterparts (in particular, this holds when $\overline{\sigma_{\Delta,A}} = \overline{\sigma_{\Delta,D}} = 0$ and $\overline{\gamma^{\uparrow}} = \overline{\gamma}$). Further, comparing the final recursions (28) and (30), full agreement of infinite-width limits requires also $\overline{\sigma_{\Delta,B}} = \overline{\sigma_{\Delta,C}} = 0$, in which case both of the terms labeled $\textcircled{1'}$ and $\textcircled{2'}$ above match $\textcircled{1}$. In this (degenerate) case where no noise is added, the training dynamics after upscaling are identical to those before upscaling, which is expected given the equivalence we showed in Section 2. Otherwise, whenever we add noise during upscaling, the infinite-width training dynamics are expected to be altered after upscaling, and no choice of hyperparameters can exactly preserve the infinite-width limit.

One may view this as an illustration that tuning hyperparameters for upscaled training is a non-trivial task distinct from tuning for ordinary training. Indeed, on the one hand and as we have discussed, we must use non-zero noise in upscaling in order for training after upscaling to yield models that are actually utilizing their increased width and learning a larger class of functions than those parameterized by narrow models. On the other hand, we see from the above calculations that, even in quite simple architectures, once we use non-zero noise, we cannot hope for upscaled training to share the infinite-width limit of its training dynamics with ordinary training. Thus, in particular, it seems that in order to tune hyperparameters for upscaled training we must, as we do in our proposed method, actually simulate upscaling itself on smaller models, rather than merely taking hyperparameters tuned on non-upscaled training and systematically modifying them in some way (unless one were to explicitly describe the limiting behavior of upscaled training in terms of that of non-upscaled training, which, even with the Ne⊗or⊤ program tools, appears to be a prohibitively complicated mathematical task).

F Experiments

F.1 MLPs

Dataset. We use the Forest Cover Type dataset (Blackard, 1998) from the UCI Machine Learning Repository, a tabular dataset for multiclass classification into 7 forest cover types based on attributes such as elevation, aspect, slope, hillshade, soil type, and additional environmental variables. The dataset comprises 581,012 samples with 54 features, including both continuous and binary variables. We apply standard preprocessing by performing a stratified 80–20 train–test split and normalizing the continuous features, while leaving the binary features unchanged. We select this dataset because MLPs achieve strong performance on tabular data of this type, and its relatively large sample size yields a sufficiently challenging task in which increasing MLP width and thus model capacity improves predictive accuracy. Therefore, it is well suited for our exploration of model upscaling.

Model. We employ a standard MLP with bias terms and ReLU activations, comprising 4 layers whose hidden width is shared and set to n. Specifically, this is MLP defined in (1) with $L = 4, n_1 = n_2 = n_3 = n$, and $\phi = \text{ReLU}$. The μP package is used to configure weight initialization and learning-rate scaling to ensure width-consistent training.

Optimizer and training. We experiment with both SGD and AdamW, and we use the μP implementations of these optimizers to obtain the appropriate scaling with respect to network width. For SGD, we apply weight decay with a base coefficient of 10^{-4} (with μP scaling) to stabilize training. For AdamW, we set $\beta = (0.9, 0.999)$, $\epsilon = 10^{-8}$, and weight decay 10^{-4}, each applied with the corresponding μP scaling. We tune the learning-rate base constant and the magnitude of added noise. We use a batch size of 2000 and train for 500 epochs.

Experiment procedure.

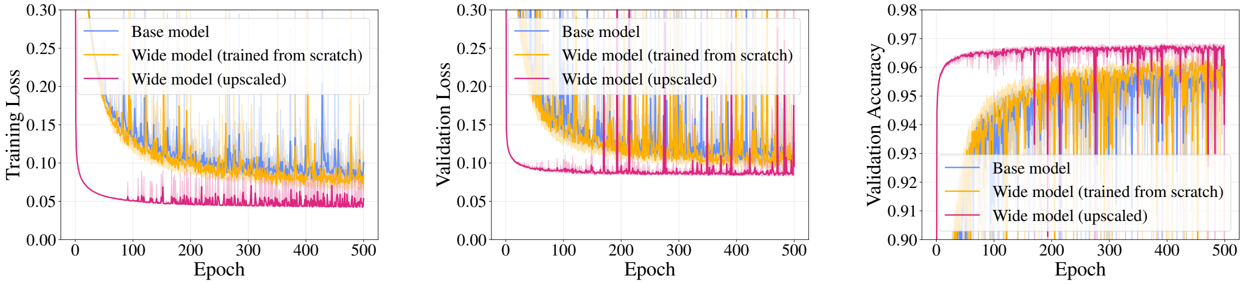
(1) Under μP, we begin with Sweep 1 over learning rates $\overline{\gamma}$ at width $n_0 = 100$, selecting the best learning rate based on the training loss after 500 epochs.

(2) We then train a base model of width $n = 500$ from scratch for 500 epochs using the best learning rate $\overline{\gamma}$ identified in Sweep 1. We also train a wide model of width $kn = 500$ from scratch using the same hyperparameters, to serve as a baseline for comparison with the upscaled model.

(3) Next, setting width multiplier $k = 4$, we select the width-n_0 checkpoint achieving the lowest training loss in Sweep 1 and perform Sweep 2 for upscaling: we construct an upscaled model of width $kn_0 = 400$, vary the noise std base constant $\overline{\sigma}$ and learning rate $\overline{\gamma^{\uparrow}}$, and train for 500 epochs.

(4) Finally, we apply the best noise level and learning rate $\overline{\gamma^{\uparrow}}$ found in Sweep 2 to upscale the base model to width $kn = 2000$ and train it for 500 epochs.

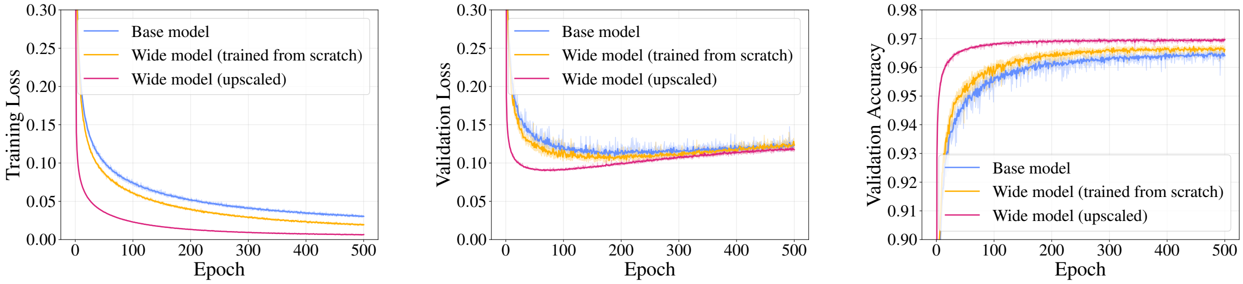
Results. Figure 4 shows training and validation curves for an MLP trained with SGD and weight decay, which are omitted from the main paper. In this setting, Sweep 1 selects a learning-rate base constant of $\overline{\gamma} = 0.4$. Sweep 2 selects a learning-rate base constant of $\overline{\gamma} = 0.1$ and a noise std base constant of $\overline{\sigma} = 5$.



(a) Training loss *(b)* Validation loss *(c)* Validation accuracy

Figure 4. Training and validation curves comparing upscaling to training from scratch for an MLP trained with SGD and weight decay. All models have width $kn = 2000$. Curves show the mean across five random runs, with ranges spanning the minimum to the maximum across runs. The y-axes are truncated to highlight differences between the two curves.

Analogous results for AdamW are shown in Figure 5, with a subset reported in the main paper. In this setting, Sweep 1 selects a learning-rate base constant of $\overline{\gamma} = 0.1$. Sweep 2 selects a learning-rate base constant of $\overline{\gamma} = 0.013$ and a noise std base constant of $\overline{\sigma} = 4$. On a side note, AdamW is more stable than SGD and generally attains higher validation accuracy on this task. Meanwhile the validation loss of the upscaled model exhibits overfitting, suggesting that the model becomes increasingly overconfident over the course of training and may be miscalibrated. Importantly, our theory of model upscaling focuses on training dynamics, specifically the behavior of the training curves. Understanding of generalization under upscaling requires a separate analytical treatment.



(a) Training loss *(b)* Validation loss *(c)* Validation accuracy

Figure 5. Training and validation curves comparing upscaling to training from scratch for an MLP trained with AdamW. All models have width $kn = 2000$. Curves show the mean across five random runs, with ranges spanning the minimum to the maximum across runs. The y-axes are truncated to highlight differences between the two curves.

F.2 ResNet

Dataset. We evaluate on CIFAR-100 (Krizhevsky et al., 2009), an image dataset of 32×32 color images with 100 classes. For training, we apply standard data augmentation comprising a 4-pixel padding followed by a random crop to 32×32 and a random horizontal flip, after which we normalize using per-channel means and standard deviations computed on the training set.

Model. We adopt the standard 18-layer ResNet from He et al. (2016) and vary its width, i.e., the number of feature channels per stage. In the standard ResNet-18, the convolutional stem outputs 64 channels, and the four residual stages use 64, 128, 256, and 512 channels, respectively. We consider width-multiplier variants: for example, the 2× model has all widths multiplying by 2, i.e. it uses 128, 256, 512, and 1024 channels. We use the μP library to configure weight initialization and learning-rate scaling to ensure width-consistent training dynamics. In particular, we configure the implementation so that the standard (1×) model exhibits identical behavior under μP and under the standard parametrization. See, e.g., Remark C.12.

Optimizer and training. Following He et al. (2016), we train with SGD using momentum 0.9 and weight decay 10^{-4}. We do not use Adam because it seems to have worse generalization behavior as compared to SGD. We tune the learning rate base constant and the magnitude of injected noise, when applicable. We use a batch size of 128 and train for 100 epochs.

Experiment procedure.

(1) Under μP, we begin with Sweep 1 over learning rates $\overline{\gamma}$ using the 0.5× models, selecting the best learning rate based on the training loss after 100 epochs.

(2) We then train a 1× base model from scratch for 100 epochs using the best learning rate $\overline{\gamma}$ identified in Sweep 1. We also train a wide 2× model from scratch using the same hyperparameters, to serve as a baseline for comparison with the upscaled model.

(3) Next, setting width multiplier $k = 2$, we select the 0.5× model checkpoint achieving the lowest training loss in Sweep 1 and perform Sweep 2 for upscaling: we construct an upscaled 1× model, vary the noise std base constant $\overline{\sigma}$ and learning rate $\overline{\gamma^{\uparrow}}$, and train for 100 epochs.

(4) Finally, we apply the best noise level and learning rate $\overline{\gamma^{\uparrow}}$ found in Sweep 2 to upscale the 1× base model to 2× and train it for 100 epochs.

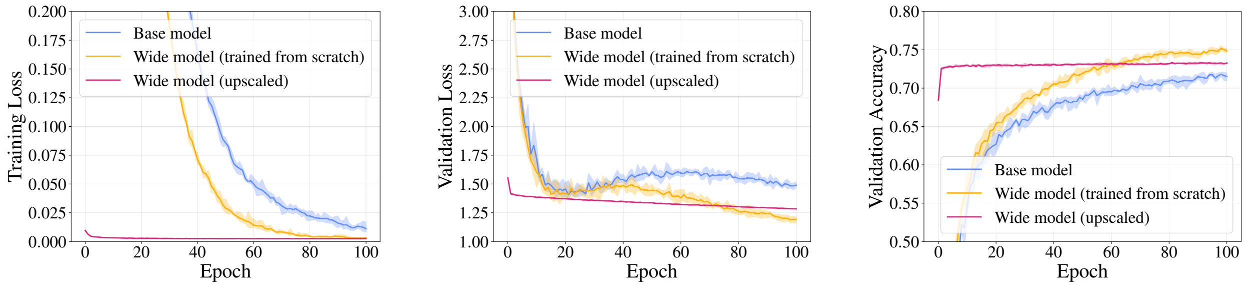
Experiment results Figure 6 shows training and validation curves for ResNets trained with SGD (using weight decay and momentum), with a subset of results reported in the main paper. In this setting, Sweep 1 selects a learning-rate base constant of $\overline{\gamma} = 0.01$. Sweep 2 selects a learning-rate base constant of $\overline{\gamma} = 0.0025$ and a noise std base constant of $\overline{\sigma} = 0.04$. For training loss, the upscaled model converges faster and attains comparable final performance; however, in validation loss and accuracy, the wide model trained from scratch outperforms the upscaled model.

In this case, Sweep 2 selects a very small symmetry-breaking noise level. These results suggest that, for this task and dataset, increasing ResNet width does not necessarily improve performance much, and a base-width model with an appropriate learning-rate schedule achieves comparable results. This likely explains why upscaling provides no benefit here.

F.3 GPT-2

Dataset. We evaluate on the `CC-MAIN-2013-20` subset[8] of the FineWeb-Edu dataset (Penedo et al., 2024). We leverage the official tokenizer of GPT-2, which has a vocabulary size of 50,257. We

[8] https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu/viewer/CC-MAIN-2013-20



(a) Training loss

(b) Validation loss

(c) Validation accuracy

Figure 6. Training and validation curves comparing upscaling to training from scratch for ResNet trained with SGD. All models are 4× ResNets. Curves show the mean across five random runs, with ranges spanning the minimum to the maximum across runs. The y-axes are truncated to highlight differences between the two curves.

construct a training split of 11.8B tokens and a validation split of 5.5M tokens.

Model. We adopt the standard GPT-2 architecture (Radford et al., 2019) and vary its hidden dimension. Specifically, we use the GPT-2 small configuration with 12 layers and 12 attention heads. We define the 4× model as GPT-2 small with 64 dimensions per head, and create the 2× and 1× models by scaling down the head dimension to 32 and 16, respectively. All models share the same number of layers and attention heads.

Optimizer and training. We utilize the AdamW optimizer with $(\beta_1, \beta_2) = (0.90, 0.95)$, using a weight decay of 0.1 and gradient clipping at 1.0. We set an effective batch size of 0.5M tokens per step and train the model for $10,000$ steps, totalling 5B tokens. A constant learning rate schedule is employed. For efficiency, we adapt mixed-precision training (bf16 with TF32-enabled `matmul`). All experiments are conducted on 2 NVIDIA H200 GPUs (141 GiB each) using PyTorch Distributed Data Parallel (DDP).

Experiment procedure.

(1) Under μP, we begin with Sweep 1 over learning rates $\overline{\gamma}$ using the 1× models, selecting the best learning rate based on end-of-training validation negative log likelihood (NLL).

(2) We then train a 2× base model from scratch for $10,000$ steps using the best learning rate $\overline{\gamma}$ identified in Sweep 1. We also train a wide 4× model from scratch using the same hyperparameters, to serve as a baseline for comparison with the upscaled model.

(3) Next, setting width multiplier $k = 2$, we select the 1× model checkpoint in Sweep 1 and perform Sweep 2 for upscaling: we construct an upscaled 2× model, vary the noise std base constant $\overline{\sigma}$ and learning rate $\overline{\gamma^{\uparrow}}$, and train for $10,000$ steps.

(4) Finally, we apply the best noise level and learning rate $\overline{\gamma^{\uparrow}}$ found in Sweep 2 to upscale the 2× base model to 4× and train it for $10,000$ steps.

F.4 Verification of hyperparameter transfer

We experimentally validate that, under Meta-Algorithm 1, hyperparameters transfer across model widths, consistent with the behavior reported in Yang et al. (2022) without upscaling.

MLP. We use the same MLP architecture, dataset, and optimizer defaults as in Appendix F.1, with the sole modification of training for 100 epochs to expedite experimentation.

For SGD with weight decay, Figure 3(b,e) reports the experiment results. Panel (b) fixes the noise std base constant at $\overline{\sigma_\Delta} = 0.75$ (near-optimal) and varies the upscaled-model learning-rate base constant $\overline{\gamma^\uparrow}$. Panel (e) fixes the learning-rate base constant at $\overline{\gamma^\uparrow} = 0.1$ and varies the amount of added noise, controlled by the noise std base constant $\overline{\sigma_\Delta}$.

Analogous results for AdamW appear in Figure 3(c,f). Panel (c) fixes $\overline{\sigma_\Delta} = 1$ and varies $\overline{\gamma^\uparrow}$. Panel (f) fixes $\overline{\gamma^\uparrow} = 0.0005$ and varies $\overline{\sigma_\Delta}$.

In all cases, the optimal hyperparameters generally transfer across widths.

GPT-2. The results for GPT-2 is reported in Figure 3(a,d). We run hyperparameter transfer verification experiments using a smaller model with 8 layers and 8 attention heads. To further control the model size, we train a BPE tokenizer with an 8,192-word vocabulary on FineWeb-edu. As in F.3, we fix the number of heads and vary only the per-head dimension. We train base models with hidden sizes $n \in \{128, 256, 512\}$ (head dimensions $\{16, 32, 64\}$), then upscale each by $k = 2$ to $N \in \{256, 512, 1024\}$ (head dimensions $\{32, 64, 128\}$). Across these settings, we sweep injected noise levels and learning rates. All base models are trained under μP with the same hyperparameters.

F.5 Further discussions

In our upscaling algorithm, we tune via hyperparameter transfer the amount of noise injected during upscaling controlled by hyperparameter $\overline{\sigma_\Delta}$. Empirically, the optimal $\overline{\sigma_\Delta}$ varies substantially with the architecture, optimizer, training horizon (number of epochs), and other factors. This motivates us to investigate the *relative* noise level with respect to the signal. To this end, we propose three alternative schemes for normalizing the added noise to have comparable "magnitude" to the signal, illustrated on an MLP.

To investigate this, we propose 3 other ways of controlling the amount of added noise (we illlustrate this with MLP example), where this time the noise is normalized so that it has similar "magnitude" to the signal.

(1) Additive, rescaled noise:

$$W^{\uparrow(\ell)} := k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top + t \, \frac{\left\| k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \right\|}{\left\| \Delta^{(\ell)} \right\|} \Delta^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}},$$

where the scaled std of the added noise is set to be $\overline{\sigma_\Delta} = 1$, and $\|\cdot\|$ denotes the spectral norm. The rescaling of the noise ensures that the signal and noise terms have equal spectral norms. Compared with the default scheme, sweeping t modulates the noise level relative to the signal.

(2) Elementwise masked noise:

$$W^{\uparrow(\ell)} := k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top + t \left| W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \right| \odot \Delta^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}.$$

where $\Delta^{(\ell)}$ has i.i.d. entries sampled from $\mathcal{N}(0,1)$, $\odot$ denotes the Hadamard product, and $|\cdot|$ denotes the entrywise absolute value. This formulation provides an alternative means of tuning the noise level relative to the signal. In this case, the noise is normalized on an entrywise basis with respect to the signal.

(3) Interpolation between signal and noise:

$$W^{\uparrow(\ell)} := \sqrt{1-t}\, k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top + \sqrt{t}\, \frac{\left\| k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \right\|}{\left\| \Delta^{(\ell)} \right\|}\, \Delta^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}.$$

where the scaled std of the added noise is set to be $\overline{\sigma_\Delta} = 1$. This construction yields $\|W^{\uparrow(\ell)}\| \approx \|k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top\|$, thereby approximately preserving the spectral norm of the weights and, consequently, the magnitude of the preactivations. We sweep $t \in \{0, 0.5, \ldots, 1\}$. When $t = 0$, the procedure produces a wider, equivalent MLP, so that the upscaled models behave as if finetuning the base model directly. When $t = 1$, it reduces to training the wide model from scratch, with a random initialization. Choosing $t \in (0, 1)$ yields an interpolation between these two regimes, with noise added to the signal.

We sweep t directly instead of $\overline{\sigma}$. In schemes (1) and (2), the optimal t typically lies in $(0, 1)$, and in scheme (3) it typically lies in $[0, 0.5]$. These observations indicate that the optimal relative noise level is generally slightly smaller than the signal, as expected.

The hyperparameter t does not necessarily transfer reliably across widths, so tuning t on small upscaling systems is not a viable extension of our hyperparameter transfer scheme. As a workaround, using scheme (1), one may tune t on a small upscaling system and then compute the effective per-layer noise base constant $\overline{\sigma_\Delta^{(\ell)}} = t\, \frac{\left\| k_{\ell-1}^{-1} W^{(\ell)} \otimes \mathbf{1}_{k_\ell} \mathbf{1}_{k_{\ell-1}}^\top \right\|}{\left\| \Delta^{(\ell)} \right\|}$. Since $\overline{\sigma_\Delta^{(\ell)}}$ is expected to transfer across widths, these values can be recorded and used for actual upscaling. This approach may potentially address the difficulty of prescribing a search grid for the best $\overline{\sigma_\Delta}$, which can vary substantially across settings. However, it entails adding a different amount of noise to each weight in practice, making implementation slightly more complex. We leave a thorough investigation of this strategy to future work.

G Conclusions and limitations

We propose a principled, theory-grounded upscaling algorithm and provide an efficient method for hyperparameter tuning, addressing a key gap in prior work. Our claim of hyperparameter transfer is anchored in μP, which was rigorously proven to have optimal training dynamics in the infinite-width limit but does not guarantee hyperparameter transfer. To date, formal proofs of such optimality exist only in very simple settings (Hayou, 2025). Thus, while our experiments show that hyperparameter transfer is effective in practice, it may not be uniformly robust across tasks and architectures. Beyond these theoretical caveats, our scope is restricted to width upscaling, whereas practical deployments often require joint increases in width and depth. Extending our framework to simultaneous width and depth upscaling is promising, and we leave this to future work. A further practical consideration concerns the tuning dimension. Our approach is most efficient when the tuning width n_0 is much smaller than the base width n, which keeps additional tuning costs modest. However, effective transfer empirically requires n_0 to be sufficiently wide (often exceeding 100 units or channels) to enter the regime where μP's asymptotic behavior is a good approximation. This tension makes the method most advantageous at large widths, yet running extremely wide experiments is infeasible for us. Finally, our analysis focuses on training dynamics rather than generalization, optimization landscape, or implicit bias. Consequently, performance may degrade in settings prone to overfitting, and understanding generalization under upscaling remains an open question.