Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer

Greg Yang[*,×] **Edward J. Hu**[*,×,†] **Igor Babuschkin**[◦] **Szymon Sidor**[◦] **Xiaodong Liu**[×]
David Farhi[◦] **Nick Ryder**[◦] **Jakub Pachocki**[◦] **Weizhu Chen**[×] **Jianfeng Gao**[×]
[×]Microsoft Corporation [◦]OpenAI

Abstract

Hyperparameter (HP) tuning in deep learning is an expensive process, prohibitively so for neural networks (NNs) with billions of parameters. We show that, in the recently discovered Maximal Update Parametrization (μP), many optimal HPs remain stable even as model size changes. This leads to a new HP tuning paradigm we call μ*Transfer*: parametrize the target model in μP, tune the HP indirectly on a smaller model, and *zero-shot transfer* them to the full-sized model, i.e., without directly tuning the latter at all. We verify μTransfer on Transformer and ResNet. For example, 1) by transferring pretraining HPs from a model of 13M parameters, we outperform published numbers of BERT-large (350M parameters), with a total tuning cost equivalent to pretraining BERT-large once; 2) by transferring from 40M parameters, we outperform published numbers of the 6.7B GPT-3 model, with tuning cost only 7% of total pretraining cost. A Pytorch implementation of our technique can be found at `github.com/microsoft/mup` and installable via `pip install mup`.

1 Introduction

Hyperparameter (HP) tuning is critical to deep learning. Poorly chosen HPs result in subpar performance and training instability. Many published baselines are hard to compare to one another due to varying degrees of HP tuning. These issues are exacerbated when training extremely large deep learning models, since state-of-the-art networks with billions of parameters become prohibitively expensive to tune.

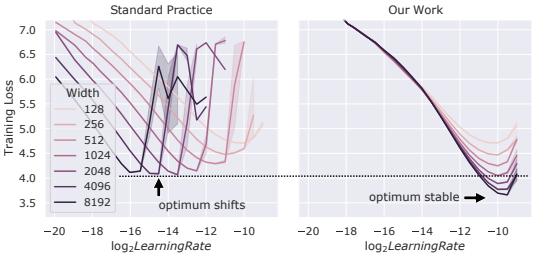


Figure 1: Training loss against learning rate on Transformers of varying d_{model} trained with Adam. Conventionally and in contrast with our technique, different widths do not share the same optimal hyperparameter; wider networks do not always perform better than narrower ones; in fact they underperform the same-width networks in our technique even after tuning learning rate (see dashed line). See Sections 3 and 4 for experimental setup.

Recently, [54] showed that different neural network parametrizations induce different infinite-width limits and proposed the *Maximal Update Parametrization (abbreviated μP)* (summarized in Table 3) that enables "maximal" feature learning in the limit. Intuitively, it ensures that each layer is updated on the same order during training *regardless of width*.[2] In contrast, while the standard parametrization (SP) ensures activations are of unit order at initialization, it actually causes them to blow up in wide models during training [54] essentially due to an imbalance of per-layer

[†]Work done partly during Microsoft AI Residency Program.
[*]Equal contribution. Order is random. Correspondence to `{gregyang, edwardhu}@microsoft.com`
[2]i.e., the updates' effect on activations becomes roughly independent of width in the large width limit.

Algorithm 1 Tuning a Large Target Model via μTransfer

1: Parametrize target model in Maximal Update Parametrization (μP)
2: Tune a smaller version (in width and/or depth) of target model
3: Copy tuned hyperparameters to target model

Table 1: **Hyperparameters That Can Be μTransferred, Not μTransferred, or μTransferred Across,** with a few caveats discussed in Section 6.1. * means *empirically validated only* on Transformers, while all others additionally have theoretical justification.

μTransferable	Not μTransferable	μTransferred *Across*
optimization related, init, parameter multipliers, etc	regularization (dropout, weight decay, etc)	width, depth*, batch size*, training time*, seq length*

learning rate (also see Fig. 5). We leverage μP to *zero-shot transfer HPs from small models to large models* in this work – that is, we obtain near optimal HPs on a large model without directly tuning it at all! While practitioners have always guessed HPs of large models from those of small models, the results are hit-or-miss at best because of incorrect parametrization. For example, as shown in Fig. 1, in a Transformer, the optimal learning rate is stable with width in μP (right) but far from so in standard parametrization (left). In addition to width, we empirically verify that, with a few caveats, HPs can also be transferred across depth (in Section 6.1) as well as batch size, language model sequence length, and training time (in Appendix G.2.1). This reduces the tuning problem of an (arbitrarily) large model to that of a (fixed-sized) small model. Our overall procedure, which we call μ*Transfer*, is summarized in Algorithm 1 and Fig. 2, and the HPs we cover are summarized in Tables 1 and 2.

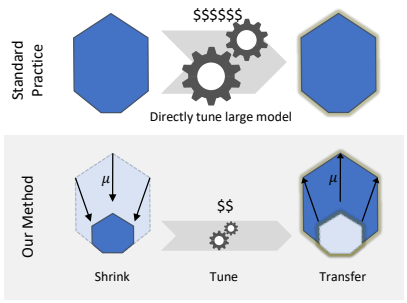


Figure 2: Illustration of μTransfer

There are several benefits to our approach: 1. **Better Performance:** μTransfer is not just about predicting how the optimal learning rate scales in SP. In general, we expect the μTransferred model to outperform its SP counterpart with learning rate optimally tuned. For example, this is the case in Fig. 1 with the width-8192 Transformer. We discuss the reason for this in Section 5 and Appendix C. 2. **Speedup:** It provides massive speedup to the tuning of large models. For example, we are able to outperform published numbers of (350M) BERT-large [11] purely by zero-shot HP transfer, with tuning cost approximately equal to 1 BERT-large pretraining. Likewise, we outperform the published numbers of the 6.7B GPT-3 model [7] with tuning cost being only 7% of total pretraining cost. For models on this scale, HP tuning is not feasible at all without our approach. 3. **Tune Once for Whole Family:** For any fixed family of models with varying width and depth (such as the BERT family or the GPT-3 family), we only need to tune a single small model and can reuse its HPs for all models in the family.[3] For example, we will use this technique to tune BERT-base (110M parameters) and BERT-large (350M parameters) simultaneously by transferring from a 13M model. 4. **Better Compute Utilization:** While large model training needs to be distributed across many GPUs, the small model tuning can happen on individual GPUs, greatly increasing the level of parallelism for tuning (and in the context of organizational compute clusters, better scheduling and utilization ratio). 5. **Painless Transition from Exploration to Scaling Up:** Often, researchers explore new ideas on small models but, when scaling up, find their HPs optimized during exploration work poorly on large models. μTransfer would solve this problem.

In addition to the HP stability property, we find that *wider is better throughout training* in μP, in contrast to SP (Section 8). This increases the reliability of model scaling in deep learning.

In this work, we primarily focus on hyperparameter transfer with respect to training loss. In settings where regularization is not the bottleneck to test performance, as in all of our experiments here, this also translates to efficacy in terms of test loss. In other settings, such as finetuning of models on small datasets, μTransfer may not be sufficient, as we discuss in Section 6.1.

[3] but possibly *not* for different data and/or tasks.

Table 2: **Examples of μTransferable Hyperparameters.** All of the below can also be specialized to per-layer hyperparameters.

Optimizer Related	Initialization	Parameter Multipliers
learning rate (LR), momentum, Adam beta, LR schedule, etc	per-layer init. variance	multiplicative constants after weight/biases, etc

Our Contributions

- We demonstrate it is possible to zero-shot transfer near optimal HPs to a large model from a small version via the Maximal Update Parametrization (μP) from [54].
- While [54] only covered SGD, here we derive μP for Adam as well (Table 3).
- We propose a new HP tuning technique, *μTransfer*, for large neural networks based on this observation that provides massive speedup over conventional methods and covers both SGD and Adam training;
- We thoroughly verify our method on machine translation and large language model pretraining (in Section 7.3) as well as image classification (in Appendix G.1);
- We release a PyTorch [33] package for implementing μTransfer painlessly. A sketch of this package is given in Appendix H.

Terminologies Sometimes, to be less ambiguous, we often refer to the "large model" as the *target model*, as it is the model we wish to ultimately tune, while we refer to the "small model" as the *proxy model*, as it proxies the HP tuning process. We follow standard notation $d_{model}, d_{head} = d_k, d_v, n_{head}, d_{ffn}$ regarding dimensions in a Transformer; one can see Fig. 11 for a refresher.

***Tensor Programs* Series** This paper is the 5th installment of the *Tensor Programs* series. While it is self-contained with the target audience being practitioners and empirical researchers, this paper presents the first major *practical* payoff of the *theoretical* foundation built in previous works [50–55].

2 Parametrization Matters: A Primer

In this section, we give a very basic primer on why the correct parametrization can allow HP transfer across width, but see Appendices J.1 to J.3 for more (mathematical) details.

The Central Limit Theorem (CLT) says that, if $x_1, \dots, x_n$ are iid samples from a zero-mean, unit-variance distribution, then $\frac{1}{\sqrt{n}}(x_1 + \cdots + x_n)$ converges to a standard Gaussian $\mathcal{N}(0, 1)$ as $n \to \infty$. Therefore, we can say that $\frac{1}{\sqrt{n}}$ is the right order of *scaling factor* c_n such that $c_n(x_1 + \cdots + x_n)$ converges to something nontrivial. In contrast, if we set $c_n = 1/n$, then $c_n(x_1 + \cdots + x_n) \to 0$; or if $c_n = 1$, then $c_n(x_1 + \cdots + x_n)$ blows up in variance as $n \to \infty$.

Now suppose we would like to minimize the function

$$F_n(c) \overset{\text{def}}{=} \mathop{\mathbb{E}}_{x_1,\dots,x_n} f(c(x_1 + \cdots + x_n)) \tag{1}$$

over $c \in \mathbb{R}$, for some bounded continuous function $f : \mathbb{R} \to \mathbb{R}$. If we reparametrize $c = \alpha/\sqrt{n}$ for $\alpha \in \mathbb{R}$, then by CLT, $G_n(\alpha) \overset{\text{def}}{=} F_n(c) \to \mathbb{E} f(\mathcal{N}(0, \alpha^2))$ stabilizes into a function of α as $n \to \infty$. Then for sufficiently large n, the optimal $\alpha_n^* \overset{\text{def}}{=} \arg\min_\alpha G_n(\alpha)$ should be close to α_N^* for any $N > n$, and indeed, for $N = \infty$ — this precisely means we can *transfer* the optimal c_n^* or α_n^* for a smaller problem (say F_n) to a larger problem (say F_N): G_N is approximately minimized by α_n^* and F_N is approximately minimized by $c_n^*\sqrt{n/N}$. Because the transfer algorithm is simply copying α, we say the parametrization $c = \alpha/\sqrt{n}$ is the *correct parametrization* for this problem.

In the scenario studied in this paper, $x_1, \dots, x_n$ are akin to randomly initialized parameters of a width-n neural network, c is akin to a HP such as learning rate, and f is the test-set performance of the network *after training*, so that F_n gives its expectation over random initializations. Just as in this example, if we parametrize the learning rate and other HPs correctly, then we can directly copy the optimal HPs for a narrower network into a wide network and expect approximately optimal

performance — this is the *(zero-shot) hyperparameter transfer* we propose here. It turns out the Maximal Update Parametrization (μP) introduced in [54] is correct (akin to the parametrization in α above), while the standard parametrization (SP) is incorrect (akin to the parametrization in c). We will review both parametrizations shortly. Theoretically, a μP network has a well-defined infinite-width limit — akin to $(x_1 + \cdots + x_n)/\sqrt{n}$ having a $\mathcal{N}(0, 1)$ limit by CLT — while a SP network does not (the limit will blow up) [54].[4] In fact, based on the theoretical foundation laid in [54], we argue in Appendix J.3 that μP should also be the *unique* parametrization that allows HP transfer across width. For a more formal discussion of the terminologies *parametrization* and *transfer*, see Appendix A

We emphasize that, to ensure transferability of any hyperparameter (such as learning rate), it's not sufficient to reparametrize *only* that hyperparameter, but rather, we need to identify and correctly reparametrize *all* hyperparameters in Table 2. For example, in Fig. 1, the wide models in SP still underperform their counterparts in μP, even with learning rate tuned optimally. This is precisely because SP does not scale parameter multipliers and input/output layer learning rates correctly in contrast to μP (see Table 3). See Appendix C for more intuition via a continuation of our example here. We shall also explain this more concretely in the context of neural networks in Section 5.

3 Hyperparameters Don't Transfer Conventionally

In the community there seem to be conflicting assumptions about HP stability. *A priori*, models of different sizes don't have any reason to share the optimal HPs. Indeed, papers aiming for state-of-the-art results often tune them separately. On the other hand, a nontrivial fraction of papers in deep learning fixes all HPs when comparing against baselines, which reflects an assumption that the optimal HPs should be stable — not only among the same model of different sizes but also among models of different designs — therefore, such comparisons are fair. Here, we demonstrate HP *instability* across width explicitly in MLP and Transformers in the standard parametrization. We will only look at training loss to exclude the effect of regularization.

MLP with Standard Parametrization We start with a 2-hidden-layer MLP with activation function ϕ, using the standard parametrization[5] with LeCun initialization[6] akin to the default in PyTorch:

$$f(\xi) = W^{3\top}\phi(W^{2\top}\phi(W^{1\top}\xi + b^1) + b^2)$$
$$\text{with init.} \quad W^1 \sim \mathcal{N}(0, 1/d_{in}),\ W^{\{2,3\}} \sim \mathcal{N}(0, 1/n),\ b^{\{1,2\}} = 0, \tag{2}$$

where $W^1 \in \mathbb{R}^{d_{in}\times n}, b^1 \in \mathbb{R}^n$, $W^2 \in \mathbb{R}^{n\times n}, b^2 \in \mathbb{R}^n$, $W^3 \in \mathbb{R}^{n\times d_{out}}$ and d_{in}, n, and d_{out} are the input, hidden, and output dimensions. The particular MLP we use has $\phi = ReLU$ and a cross-entropy (xent) loss function. We define the width of MLP as the hidden size n, which is varied from 256 to 8192. The models are trained on CIFAR-10 for 20 epochs, which is more than enough to ensure convergence.

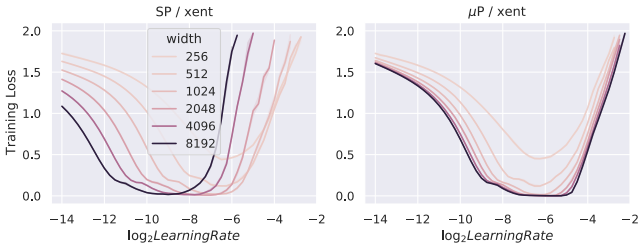


Figure 3: MLP width different hidden sizes trained for 20 epoch on CIFAR-10 using SGD. **Left** uses standard parametrization (SP); **right** uses maximal update parametrization (μP). μP networks exhibit better learning rate stability than their SP counterparts.

As shown on the left in Fig. 3, the optimal learning rate shifts by roughly an order of magnitude as the width increases from 256 to 8192; using the optimal learning of the smallest model on the largest model gives very bad performance, if not divergence.

Transformer with Standard Parametrization This perhaps unsurprising observation holds for more complex architectures such as Transformer as well, as shown in Fig. 1 (left). We define width

[4] The more theoretically astute reader may observe that SP with a $\Theta(1/width)$ learning rate induces a well-defined infinite-width limit exists as well. Nevertheless, this does not allow HP transfer because this limit is in kernel regime as shown in [54]. See Appendix J.3 for more discussions.

[5] i.e. the default parametrization offered by common deep learning frameworks. See Table 3 for a review.

[6] The key here is that the init. variance $\propto 1/\text{fan\_in}$, so the same insights here apply with e.g. He initialization.

Table 3: **μP[54] and SP for General Neural Networks.** Here, we emphasize the *scaling with width* (fan_in *or* fan_out); in practice, we may insert tunable multipliers in front of fan_in and fan_out as in Eq. (4). The fan_out of a bias vector is its dimension (whereas fan_in is 1). Purple text highlights key differences from standard parametrization (SP); Gray text recalls the corresponding SP. *SGD* (resp. *Adam*) here can be replaced by variants such as SGD with momentum (resp. Adagrad, etc); see Appendix B.3 for other optimizers. In general, the three columns here can be interpreted as linear layers that have {finite, infinite, infinite} input dimension and {infinite, finite, infinite} output dimension in an infinite-width network; this description generalizes more readily to other parameters such as those of layernorm. Transformer μP requires one more modification ($1/d$ attention instead of $1/\sqrt{d}$); see Definition 4.1. This version of μP gets rid of parameter multipliers; for the version similar to that in [54], see Table 9. Also see Table 8 for a μP formulation that is easier to implement (and compatible with input/output weight sharing). Further explanation of this table can be found in Appendix B. Its derivation can be found in Appendix J.

	Input weights & all biases	Output weights	Hidden weights
Init. Var.	$1/\text{fan\_in}$	$1/\text{fan\_in}^2$ $(1/\text{fan\_in})$	$1/\text{fan\_in}$
SGD LR	fan_out (1)	$1/\text{fan\_in}$ (1)	1
Adam LR	1	$1/\text{fan\_in}$ (1)	$1/\text{fan\_in}$ (1)

as d_{model}, with $d_k = d_q = d_v = {}^{d_{model}}/_{n_{head}}$ and $d_{ffn} = 4d_{model}$. The models are trained on wikitext-2 for 5 epochs. In Fig. 18 in the appendix we also show the instability of initialization scale and other HPs.

4 Unlocking Zero-Shot Hyperparameter Transfer with μP

We show that μP solves the problems we see in Section 3.

MLP with μP For the MLP in Section 3, to switch to μP, we just need to modify Eq. (2)'s initialization of the last layer and its learning rates of the first and last layer as well as of the biases. The *basic form* is[7]

$$\begin{aligned} &\text{initialize} \quad W^1 \sim \mathcal{N}(0, {}^1/_{d_{in}}),\ W^2 \sim \mathcal{N}(0, {}^1/_{n}),\ W^3 \sim \mathcal{N}(0, {}^1/_{n^2}),\ b^{\{1,2\}} = 0 \\ &\text{with SGD learning rates} \quad \eta_{W^1} = \eta_{b^1} = \eta_{b^2} = \eta n,\ \eta_{W^2} = \eta,\ \eta_{W^3} = \eta n^{-1}. \end{aligned} \tag{3}$$

Here, η specifies the "master" learning rate, and we highlighted in purple the differences in the two parametrizations. This basic form makes clear the *scaling with width* n of the parametrization, but in practice we will often insert (possibly tune-able) multiplicative constants in front of each appearance of n. For example, this is useful when we would like to be consistent with a SP MLP at a *base width* n_0. Then we may insert constants as follows: For $\tilde{n} \overset{\text{def}}{=} n/n_0$,

$$\begin{aligned} &\text{initialize} \quad W^1 \sim \mathcal{N}(0, {}^1/_{d_{in}}),\ W^2 \sim \mathcal{N}(0, {}^1/_{n}),\ W^3 \sim \mathcal{N}(0, {}^1/_{n\cdot\tilde{n}}),\ b^{\{1,2\}} = 0 \\ &\text{with SGD learning rates} \quad \eta_{W^1} = \eta_{b^1} = \eta_{b^2} = \eta\tilde{n},\ \eta_{W^2} = \eta,\ \eta_{W^3} = \eta\tilde{n}^{-1}. \end{aligned} \tag{4}$$

Then at width $n = n_0$, all purple factors above are 1, and the parametrization is identical to SP (Eq. (2)) at width n_0. Of course, as n increases from n_0, then Eq. (4) quickly deviates from Eq. (2). In other words, for a particular n, μP and SP can be identical up to the choice of some constants (in this case n_0), but μP determines a different "set" of networks and optimization trajectory than SP as one varies n. As we will see empirically in the next section, this deviation is crucial for HP transfer.

Indeed, in Fig. 3(right), we plot the CIFAR10 performances, over various learning rates and widths, of μP MLPs with $n_0 = 128$. In contrast to SP, the optimal learning rate under μP is stable. This means that, the best learning rate for a width-128 network is also best for a width-8192 network in μP — i.e. HP transfer *works* — but not for SP. In addition, we observe performance for a fixed learning rate always weakly improves with width in μP , but not in SP.

This MLP μP example can be generalized easily to general neural networks trained under SGD or Adam, as summarized in Table 3, which is derived in Appendix J.

[7] While superficially different, this parametrization is equivalent to the μP defined in [54].

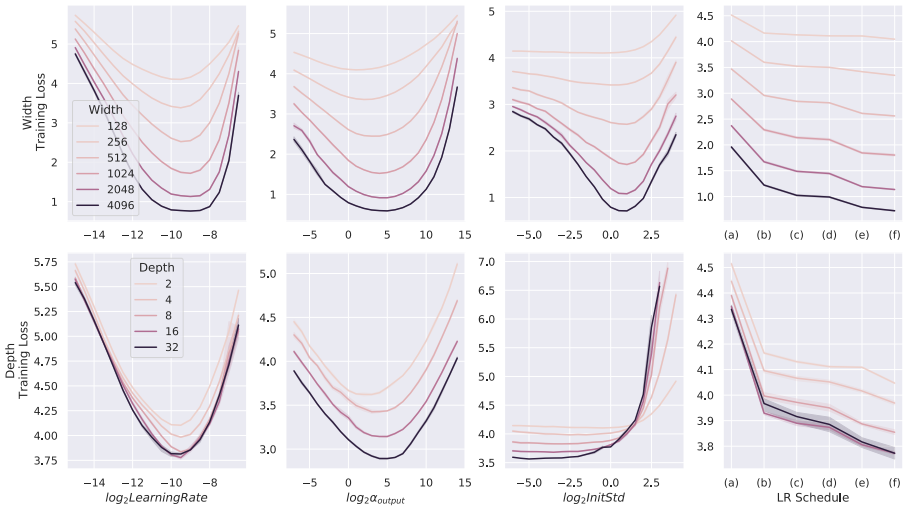


Figure 4: **Empirical validation of the stability of four representative hyperparameters on pre-LN Transformers in μP**: learning rate, last layer weight multiplier α_{output}, weight initialization standard deviation, and learning rate schedule. We use the following learning rate schedules: (a) linear decay; (b) StepLR @ [5k, 8k] with a decay factor of 0.1; (c) StepLR @ [4k, 7k] with a decay factor of 0.3; (d) cosine annealing; (e) constant; (f) inverse square-root decay. All models are trained on wikitext-2 for 10k steps. When not specified in the legend, the width used is 256, depth 2, batch size 20, sequence length 256, and LR schedule constant. We sweep a particular HP, corresponding to each column, while fixing all others constant. See Section 6.1 for discussion of these results.

Transformers with μP We repeat the experiments with base width $n_0 = 128$ for Transformers:

Definition 4.1. The *Maximal Update Parametrization (μP) for a Transformer* is given by Table 3 and $1/d$ attention instead of $1/\sqrt{d}$, i.e. the attention logit is calculated as $q^\top k/d$ instead of $q^\top k/\sqrt{d}$ where query q and key k have dimension d.[8]

The results are shown on the right in Fig. 1, where the optimal learning rate is stable, and the performance improves monotonically with width. See Appendix B for further explanation of μP.

5 The Defects of SP and How μP Fixes Them

The question of SP vs μP has already been studied at length in [54]. Here we aim to recapitulate the key insights, with more explanations given in Appendix J.3.

An Instructive Example As shown in [54] and Appendix J.3, in SP, the network output will blow up with width after 1 step of SGD. It's instructive to consider a 1-hidden-layer linear perceptron $f(x) = V^\top U x$ with scalar inputs and outputs, as well as weights $V, U \in \mathbb{R}^{n\times 1}$. In SP, $V_\alpha \sim \mathcal{N}(0, 1/n)$ ad $U_\alpha \sim \mathcal{N}(0, 1)$ for each $\alpha \in [n]$. This sampling ensures that $f(x) = \Theta(|x|)$ at initialization. After 1 step of SGD with learning rate 1, the new weights are $V' \leftarrow V + \theta U, U' \leftarrow U + \theta V$, where θ is some scalar of size $\Theta(1)$ depending on the inputs, labels, and loss function. But now

$$f(x) = V'^\top U' x = (V^\top U + \theta U^\top U + \theta V^\top V + \theta^2 U^\top V)x \tag{5}$$

blows up with width n because $U^\top U = \Theta(n)$ by Law of Large Numbers.

Now consider the same network in μP. According to Table 3, we now have $V_\alpha \sim \mathcal{N}(0, 1/n^2)$ in contrast to SP, but $U_\alpha \sim \mathcal{N}(0, 1)$ as before, with learning rates $\eta_V = 1/n, \eta_U = n$. After 1 step of SGD, we now have

$$f(x) = (V^\top U + \theta n^{-1} U^\top U + \theta n V^\top V + \theta^2 U^\top V)x,$$

[8] This is roughly because during training, q and k will be correlated so $q^\top k$ actually scales like d due to Law of Large Numbers, in contrast to the original motivation that q, k are uncorrelated at initialization so Central Limit applies instead. See Appendix J.2.1 for a more in-depth discussion.

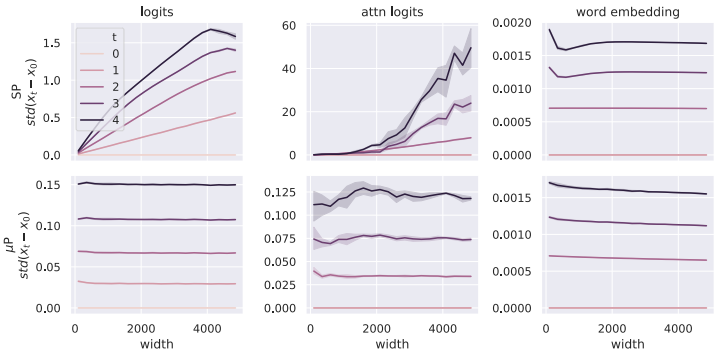


Figure 5: **Logits and attention logits, but not word embeddings, of a Transformer blow up with width in SP after 1 step of training.** In contrast, all three are well-behaved with width in μP. Here we measure how much different values change coordinatewise from initialization over 4 steps of Adam updates, as a function of width. Specifically, we plot the standard deviation of the coordinates of $x_t - x_0$, for $t = 0, \ldots, 4$, and $x \in \{\text{logits, attention logits, word embeddings}\}$, where $t = 0$ indicates initialization.

and one can verify this is $\Theta(1)$ and thus does not blow up with width.[9]

Some Layers Update Too Fast, Others Too Slow One can observe the same behavior in more advanced architectures like Transformers and optimizers like Adam; in fact, in SP, other hidden quantities like attention logits will also blow up with width after 1 step, but in μP still remain bounded, as shown in Fig. 5(middle).

One might think scaling down the learning rate with width can solve this problem in SP. However, other hidden activations like the word embedding (Fig. 5(right)) in a Transformer update by a width-independent amount for each step of training, so scaling down the learning rate will effectively mean the word embeddings are not learned in large width models. Similar conclusions apply to other models like ResNet (in fact, one can observe in the SP linear MLP example above, the input layer is updated much more slowly than the output layer). On the other hand, μP is designed so that all hidden activations update with the same speed in terms of width (see Appendix J.2 for why).

Performance Advantage of μP This is why a wide model tuned with μTransfer should in general outperform its SP counterpart with (global) learning rate tuned. For example, this is the case for the width-8192 Transformer in Fig. 1, where, in SP, the optimal learning rate needs to mollify the blow-up in quantities like logits and attention logits, but this implies others like word embeddings do not learn appreciably. This performance advantage means μTransfer does more than just predicting the optimal learning rate of wide SP models. Relatedly, we observe, for any fixed HP combination, training performance never decreases with width in μP, in contrast to SP (e.g., the μP curves in Figs. 1, 3 and 16 do not cross, but the SP curves do; see also Section 8).

6 Which Hyperparameters Can Be μTransferred?

In this section, we explore how common HPs fit into our framework. In general, they can be divided into three kinds, summarized in Table 1:

1. those that can transfer from the small to the large model, such as learning rate (Table 2);
2. those that primarily control regularization and don't work well with our technique; and
3. those that define training *scale*, such as width as discussed above as well as others like depth and batch size, across which we transfer other HPs.

Those in the first category transfer across width, as theoretically justified above in Section 2. To push the practicality and generality of our technique, we empirically explore the transfer across

[9] Note in this example, Glorot initialization [13] (i.e. with variance $1/(\text{fan\_in} + \text{fan\_out})$) would scale asymptotically the same as μP and thus is similarly well-behaved. However, if one adds layernorm or batchnorm, then Glorot will cause logit blowup like SP, but μP still will not.

the other dimensions in the third category. Note that μTransfer across width is quite general, e.g. it allows varying width ratio of different layers or number of attention heads in a Transformer; see Appendix E.2. This will be very useful in practice. For the second category, the amount of regularization (for the purpose of controlling overfitting) naturally depends on both the model size and data size, so we should not expect transfer to work if the parametrization only depends on model size. We discuss these HPs in more detail in Appendix E.1.

6.1 Empirical Validation and Limitations

Our empirical investigations focus on Transformers (here) and ResNet (in Appendix G.1.1), the most popular backbones of deep learning models today. We train a 2-layer pre-layernorm μP[10] Transformer with 4 attention heads on Wikitext-2. We sweep one of four HPs (learning rate, output weight multiplier, initialization standard deviation, and learning rate schedule) while fixing the others and sweeping along width and depth (with additional results in Fig. 19 on transfer across batch size, sequence length, and training time). Fig. 4 shows the results averaged over 5 random seeds.

Empirically, we find that for language modeling on Transformers, HPs generally transfer across scale dimensions if some minimum width (e.g. 256), depth (e.g., 4), batch size (e.g., 32), sequence length (e.g., 128), and training steps (e.g., 5000) are met, and the target scale is within the "reasonable range" as in our experiments. Now, there are some caveats. While the exact optimum can shift slightly with increasing scale, this shift usually has very small impact on the loss, compared to SP (Figs. 1 and 3(left)). However, there are some caveats. For example, the best initialization standard deviation does not seem to transfer well across depth (2nd row, 3rd column), despite having a stabler optimum across width. In addition, while our results on width, batch size, sequence length, and training time still hold for post-layernorm (Fig. 17),[11] the transfer across depth only works for pre-layernorm Transformer. Nevertheless, in practice (e.g. our results in Section 7.3) we find that fixing initialization standard deviation while tuning other HPs works well when transferring across depth.

7 Efficiency and Performance of μTransfer

Now that the plausibility of μTransfer has been established in toy settings, we turn to more realistic scenarios to see if one can achieve tangible gains. Specifically, we perform HP tuning only on a smaller proxy model, test the obtained HPs on the large target model directly, and compare against baselines tuned using the target model. We seek to answer the question: Can μTransfer make HP tuning more efficient while achieving performance on par with traditional tuning? As we shall see by the end of the section, the answer is positive. We focus on Transformers here, while experiments on ResNets on CIFAR10 and Imagenet can be found as well in Appendix G.1. All of our experiments are run on V100 GPUs.

7.1 Transformer on IWSLT14 De-En

Setup IWSLT14 De-En is a well-known machine translation benchmark. We use the default IWSLT (post-layernorm) Transformer implemented in `fairseq` [31] with 40M parameters, which we denote as the *1x model*.[12] For μTransfer, we tune on a *0.25x model* with $1/4$ of the width, amounting to 4M parameters. For this experiment, we tune via random search the learning rate η, the output layer parameter multiplier α_{output}, and the attention key-projection weight multiplier α_{attn}. See the grid and other experimental details in Appendix F.1.

We compare transferring from the 0.25x model with tuning the 1x model while controlling the total tuning budget in FLOPs.[13] To improve the reproducibility of our result: 1) we repeat the entire HP search process (a *trial*) 25 times for each setup, with number of samples as indicated in Table 4, and report the 25th, 50th, 75th, and 100th percentiles in BLEU score; 2) we evaluate each selected HP combination using 5 random initializations and report the mean performance.[14]

[10] "2 layers" means the model has 2 self-attention blocks. To compare with SP Transformer, see Fig. 18.

[11] in fact, post-layernorm Transformers are much more sensitive to HPs than pre-layernorm, so our technique is more crucial for them, especially for transfer across width. Fig. 1 uses post-layernorm.

[12] `https://github.com/pytorch/fairseq/blob/master/examples/translation/README.md`.

[13] Ideally we would like to measure the wall clock time used for tuning. However, smaller models such as the proxy Transformer used for IWSLT are not efficient on GPUs, so wall clock time would not reflect the speedup for larger models like GPT-3. Thus, we measure in FLOPs, which is less dependent on hardware optimization.

[14] We do not report the standard deviation over random initializations to avoid confusion.

Table 4: **Transformer on IWSLT14 De-En.** 1x and 0.25x refers to scaling of width only. Compared to traditional tuning ("Tuning on 1x"), μTransfer from 0.25x provides better and more reliable outcome given fixed amount of compute. On the other hand, naive transfer (i.e. with SP instead of μP) fails completely. The percentiles are over independent trials, with each trial involving the entire tuning process with a new HP random search.

Setup	Total Compute	#Samples	Val. BLEU Percentiles 25	50	75	100
`fairseq`[31] default	-	-	-	-	-	35.40
Tuning on 1x	1x	5	33.62	35.00	35.35	35.45
Naive transfer from 0.25x	1x	64	training diverged			
μTransfer from 0.25x (Ours)	1x	64	**35.27**	**35.33**	**35.45**	**35.53**

We pick the HP combination that achieves the lowest validation loss[15] for each trial. The reported best outcome is chosen according to the validation loss during tuning. We compare against the default in `fairseq`, which is presumably heavily tuned. The result is shown in Table 4.

Performance Pareto Frontier The result above only describes a particular compute budget. Is μTransfer still preferable when we have a lot more (or less) compute? To answer this question, we produce the compute-performance Pareto frontier in Fig. 6(left), where we repeat the above experiment with different compute budgets. Evidently, our approach completely dominates conventional tuning.

Sample Quality of Proxy Model vs Target Model The Pareto frontier in Fig. 6(right) suggests that, given a fixed number of random *samples* from the HP space, 1) tuning the target model directly yields slightly better results than tuning the proxy model (while taking much more compute of course), but 2) this performance gap seems to vanish as more samples are taken. This can be explained by the intuition that the narrower proxy model is a "noisy estimator" of the wide target model [54].With few samples, this noise can distort the random HP search, but with more samples, this noise is suppressed.

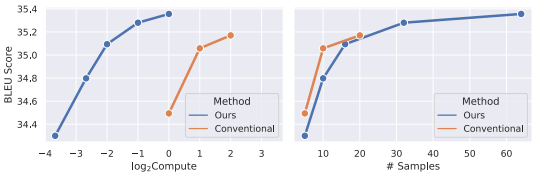


Figure 6: **Efficiency-performance Pareto frontier** of μTransfer compared to conventional tuning, on IWSLT Transformer, using random HP search as the base method. We plot the *median* BLEU score over 25 trials (Left) against relative compute budget in log scale and (Right) against number of HP samples taken. While with the same number of samples, μTransfer slightly underperforms conventional tuning, this gap vanishes with more samples, and in terms of compute, our Pareto frontier strongly and consistently dominates that of conventional tuning. Note that, in larger models (e.g. BERT or GPT-3, not shown here), we believe our efficiency advantage will only widen as our small proxy model can stay the same size while the target model grows.

7.2 Transformer on WMT14 En-De

We scale up to WMT14 En-De using the large (post-layernorm) Transformer from [47] with 211M parameters. We tune on a proxy model with 15M parameters by shrinking d_{model}, d_{ffn}, and n_{head}. For this experiment, we tune via random search the learning rate η, the output layer parameter multiplier α_{output}, and the attention key-projection weight multiplier α_{attn} following the grid in Appendix F.2. The result is shown in Table 5: While random search with 3 HP samples far underperforms the `fairseq` default, we are able to match it via transfer using the same tuning budget.

7.3 BERT

Finally, we consider large-scale language model pretraining where HP tuning is known to be challenging. Using Megatron (pre-layernorm) BERT [40] as a baseline, we hope to recover the performance of the published HPs by only tuning a proxy model that has roughly 13M parameters, which we call *BERT-prototype*. While previous experiments scaled only width, here we will also scale depth, as discussed in Section 6 and validated in Fig. 4. We use a batch size of 256 for all runs and follow the

[15] We find this provides more reliable result than selecting for the best BLEU score.

Table 5: **Transformers on WMT14 En-De.** 1x and 0.25x refers to scaling of width only. We report BLEU fluctuation over 3 independent trials, i.e., 3 independent random HP searches.

Setup	Total Compute	#Samples	Val. BLEU Percentiles Worst	Median	Best
`fairseq`[31] default	-	-	-	-	26.40
Tuning on 1x	1x	3	training diverged		25.69
Naive transfer from 0.25x	1x	64	training diverged		
μTransfer from 0.25x (Ours)	1x	64	**25.94**	**26.34**	**26.42**

standard finetuning procedures. For more details on BERT-prototype, what HPs we tune, and how we finetune the trained models, see Appendix F.3.

During HP tuning, we sample 256 combinations from the search space and train each combination on BERT-prototype for 10^5 steps. The total tuning cost measured in FLOPs is roughly the same as training 1 BERT-large for the full 10^6 steps; the exact calculation is shown in Appendix F.3. The results are shown in Table 6. Notice that on BERT-large, we obtain sizeable improvement over the well-tuned Megatron BERT-large baseline.

Table 6: **BERT pretraining.** HP transfer outperforms published baselines without tuning the full model directly at all. We tune BERT-base and BERT-large simultaneously via a single proxy model, *BERT-prototype*. The total tuning cost = the cost of pretraining a single BERT-large. *Model speedup* refers to the training speedup of BERT-prototype over BERT-base or BERT-large. *Total speedup* in addition includes time saving from transferring across training steps. Both speedups can be interpreted either as real-time speedup on V100s or as FLOPs speedup (which turn out to be empirically very similar in this case).

Model	Method	Model Speedup	Total Speedup	Test loss	MNLI (m/mm)	QQP
BERT_{base}	Megatron Default	1x	1x	1.995	84.2/84.2	90.6
BERT_{base}	Naive Transfer	4x	40x		training diverged	
BERT_{base}	μTransfer (Ours)	4x	40x	**1.970**	**84.3/84.8**	**90.8**
BERT_{large}	Megatron Default	1x	1x	1.731	86.3/86.2	90.9
BERT_{large}	Naive Transfer	22x	220x		training diverged	
BERT_{large}	μTransfer (Ours)	22x	220x	**1.683**	**87.0/86.5**	**91.4**

7.4 GPT-3

In order to further verify μTransfer at scale, we applied it to GPT-3 6.7B [7] with relative attention. This *target model* consists of 32 residual blocks with width 4096. We form the small *proxy model* by shrinking width to 256, resulting in roughly 40 million trainable parameters, 168 times smaller than the target model. HPs were then determined by a random search on the proxy model. The total tuning cost was only 7% of total pretraining cost. Details of the HP sweep can be found in Appendix F.4.

In order to exclude code difference as a possible confounder, we also re-trained GPT-3 6.7B from scratch using the original HPs from [7]. Unfortunately, after we have finished all experiments, we found this baseline mistakenly used absolute attention (like models in [7]) when it was supposed to use relative attention like the target model. In addition, during training of the μTransfer model we encountered numerical issues that lead to frequent divergences. In order to avoid them, the model was trained using FP32 precision, even though the original 6.7B model and our re-run were trained using FP16.[16] [17] The resulting μTransfer model outperforms the 6.7B from [7], and is in fact comparable to the twice-as-large 13B model across our evaluation suite (see Table 11). Selected evaluation results can be found in Table 7 and further details are given in Table 10 and Appendix F.4.

[16] While we are mainly focused on the efficacy of μTransfer regardless of precision, it would be interesting to ablate the effect of precision in our results, but we did not have enough resources to rerun the baseline in FP32

[17] It is quite interesting that μTransfer identified a useful region of hyperparameters leading to much improved performance, which probably would be difficult to discover normally because 1) researchers usually change hyperparameters to accomodate precision and 2) there was no precise enough justification to go against this judgment until μTransfer.

Table 7: **GPT-3 6.7B Pretraining.** Selected evaluation results for the GPT-3 6.7B model tuned with μTransfer (transfered from a small proxy model of 40M parameters), compared to the results published in [7] and a re-run with original HPs, as well as the 13B model in [7] for reference. Note that the perplexities in this table are based on a custom tokenization and are not comparable to the literature. The validation loss refers to the loss achieved on a random held-out part of our dataset. *Zero-shot*, *One-Shot* and *Few-Shot* refer to the number of additional query and answer pairs passed in the context when performing the sampling-based evaluations. See Appendix F.4 for full evaluation.

Task	Metric	6.7B+μP	6.7B re-run	6.7B [7]	13B [7]
Validation loss	cross-entropy	**1.98**	2.03	-	-
PTB	perplexity	**11.4**	13.0	-	-
WikiText-103	perplexity	**8.56**	9.13	-	-
One Billion Words	perplexity	**20.5**	21.7	-	-
LAMBADA Zero-Shot	accuracy	**73.5**	70.8	70.3	72.5
LAMBADA One-Shot	accuracy	**69.9**	64.8	65.4	69.0
LAMBADA Few-Shot	accuracy	74.7	77.1	**79.1**	**81.3**
HellaSwag Zero-Shot	accuracy	**72.0**	66.7	67.4	70.9
HellaSwag One-Shot	accuracy	**71.1**	65.9	66.5	70.0
HellaSwag Few-Shot	accuracy	**72.4**	66.4	67.3	71.3

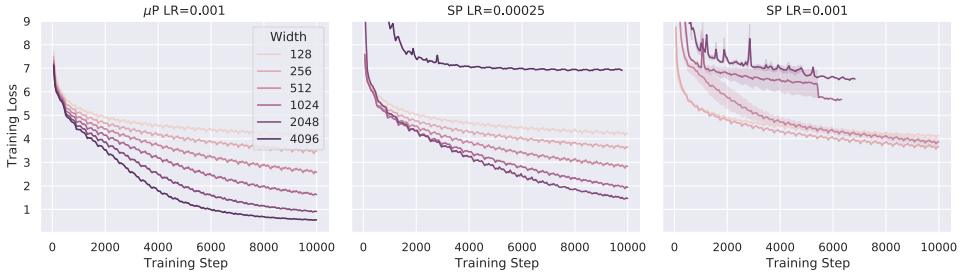


Figure 7: **Wider is always better in training loss under μP, but not in SP, given the same HP.** Learning curves for μP and SP with different learning rates, aggregated over 5 seeds. **(Left)** Wider μP models always achieve better training loss at any time in training. **(Middle)** If using a small learning rate, SP models can appear to do so up to some large width, at which point the pattern fails (at width 2048 in our plot). **(Right)** If using a large learning rate, SP model can strictly do *worse* with width; here the SP model is identical to the μP model in (Left) at width 128.

8 Wider is Better in μP *Throughout Training*

In earlier plots like Figs. 1 and 3, we saw that at the end of training, wider is always better in μP but not in SP. In fact, we find this to be true *throughout training*, as seen in Fig. 7, modulo noise from random initialization and/or data ordering, and assuming the output layer is zero-initialized (which has no impact on performance as discussed in Appendix D.2). We then stress-tested this on a μP GPT-3 Transformer (on the GPT-3 training data) by scaling width from 256 to 32,768 using a fixed set of HPs (Fig. 8). Wider models consistently match or outperform narrower models at each point in training (except a brief period around 1e8 training tokens, likely due to noise because we ran only 1 seed due to computational cost). Our observation suggests that wider models are strictly more data-efficient if scaled appropriately. By checking "wider-is-better" early in training, one can also cheaply debug a μP implementation.

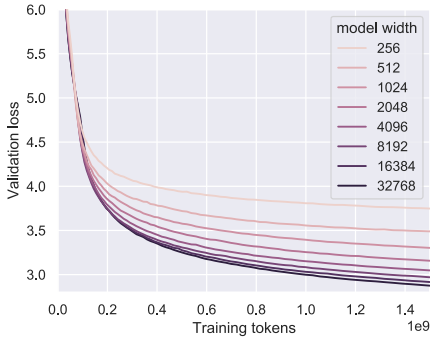


Figure 8: **Stress-testing "wider-is-better" in μP.** Here we trained a GPT-3 transformer with 4 layers and widths from 256 to 32,768. Modulo a brief period around 1e8 training tokens, wider is better throughout training.

9 *Useful* Hyperparameter Transfer: A Theoretical Puzzle

We want to tune HPs on a small model with width N such that its HP landscape looks like that of a large model with width $\gg N$. Our intuition in Section 2 and Appendices C and J leads us to μP. However, for this to be useful, we *do not want* the small model (as a function) after training to be close to that of the large model — otherwise there is no point in training the large model to begin with. So N 1) must be large enough so that the HP optimum converges, but 2) cannot be so large that the functional dynamics (and the loss) converges. The fact that such N exists, as demonstrated by our experiments, shows that: In some sense, the HP optimum is a "macroscopic" or "coarse" variable which converges quickly with width, while the neural network function (and its loss) is a very "microscopic" or "fine" detail that converges much more slowly with width. However, theoretically, it is unclear why this should happen, and where else we should expect such *useful* HP transfer. We leave an explanation to future work.

10 Related Works

10.1 Hyperparameter Tuning

Many have sought to speedup HP tuning beyond the simple grid or random search. Snoek et al. [42] treated HP tuning as an optimization process and used Bayesian optimization by treating the performance of each HP combination as a sample from a Gaussian process (GP). Snoek et al. [43] further improved the runtime by swapping the GP with a neural network. Another thread of work investigated how massively parallel infrasture can be used for efficient tuning under the multi-arm bandit problem [18, 21]. There are also dedicated tools such as Optuna [4] and Talos [3] which integrate with existing deep learning frameworks and provide an easy way to apply more advanced tuning techniques.

Our approach is distinct from all of the above in that it does not work on the HP optimization process itself. Instead, it decouples the size of the target model from the tuning cost, which was not feasible prior to this work. This means that **no matter how large the target model is, we can always use a fixed-sized proxy model to probe its HP landscape**. Nevertheless, our method is complementary, as the above approaches can naturally be applied to the tuning of the proxy model; it is only for scientific reasons that we use either grid search or random search throughout this work.

10.2 Hyperparameter Transfer

Many previous works explored transfer learning of HP tuning (e.g. [15, 34, 44, 59]). However, to the best of our knowledge, our work is the first to explore *zero-shot* HP transfer. In addition, we focus on transferring across model scale rather than between different tasks or datasets. Some algorithms like Hyperband [22] can leverage cheap estimates of HP evaluations (like using a small model to proxy a large model) but they are not zero-shot algorithms, so would still be very expensive to apply to large model training. Nevertheless, all of the above methods are complementary to ours as they can be applied to the tuning of our proxy model.

10.3 Previously Proposed Scaling Rules of Hyperparameters

(Learning Rate, Batch Size) Scaling [41] proposed to scale learning rate with batch size while fixing the total epochs of training; [14] proposed to scale learning rate as $\sqrt{batchsize}$ while fixing the total number of steps of training. However, [38] showed that there's no consistent (learning rate, batch size) scaling law across a range of dataset and models. Later, [28] studied the trade-off of training steps vs computation as a result of changing batch size. They proposed an equation of $a/(1 + b/batchsize)$, where a and b are task- and model-specific constants, for the optimal learning rate (see their fig 3 and fig 5). This law suggests that for sufficiently large batch size, the optimal learning rate is roughly constant.[18] This supports our results here as well as the empirical results in [38, fig 8].

Learning Rate Scaling with Width Assuming that the optimal learning rate should scale with batch size following [41], [32] empirically investigated how the optimal "noise ratio" $LR/batchsize$ scales with width for MLP and CNNs in NTK parametrization (NTP) or standard parametrization

[18] while the optimal learning is roughly linear in batch size when the latter is small

(SP) trained with SGD. They in particular focus on test loss in the regime of small batch size and training to convergence. In this regime, they claimed that in networks without batch normalization, the optimal noise ratio is constant in SP but scales like $1/width$ for NTP. However, they found this law breaks down for networks with normalization.

In contrast, here we focus on training loss, without training to convergence and with a range of batch sizes from small to very large (as is typical in large scale pretraining). Additionally, our work applies universally to 1) networks with normalization, along with 2) Adam and other adaptive optimizers; furthermore 3) we empirically validate transfer across depth and sequence length, and 4) explicitly validate tuning via μTransfer on large models like BERT-large and GPT-3.

Finally, as argued in [54] and Appendix J.3, SP and NTP lead to bad infinite-width limits in contrast to μP and hence are suboptimal for wide neural networks. For example, sufficiently wide neural networks in SP and NTP would lose the ability to learn features, as concretely demonstrated on word2vec in [54].

Input Layer Parametrization The original formulation of μP in [54] (see Table 9, which is equivalent to Table 3) uses a fan-out initialization for the input layer. This is atypical in vision models, but in language models where the input and output layers are shared (corresponding to word embeddings), it can actually be more natural to use a fan-out initialization (corresponding to fan-in initialization of the output layer). In fact, we found that `fairseq` [31] by default actually implements both the fan-out initialization and the $\sqrt{\text{fan\_out}}$ multiplier.

Other Scaling Rules Many previous works proposed different initialization or parametrizations with favorable properties, such as better stability for training deep neural networks [5, 13, 16, 25, 37, 56, 57, 63]. Our work differs from these in that we focus on the transferability of optimal HPs from small models to large models in the same parametrization.

10.4 Infinite-Width Neural Networks: From Theory to Practice and Back

[54] introduced μP as the unique parametrization that enables all layers of a neural network to learn features in the infinite-width limit, especially in contrast to the NTK parametrization [17] (which gives rise to the NTK limit) that does not learn features in the limit. Based on this theoretical insight, in Appendix J.3, we argue that μP should also be the *unique* parametrization (in the sense of [54]) that allows HP transfer across width; in short this is because it both 1) preserves feature learning, so that performance on feature learning tasks (such as language model pretraining) does not become trivial in the limit, and 2) ensures each parameter tensor is not stuck at initialization in the large width limit, so that its learning rate does not become meaningless. At the same time, our results here suggest that μP is indeed the *correct* parametrization for wide neural networks and thus provide empirical motivation for the theoretical study of the infinite-width μP limit. Note, *parametrization* here refers to a rule to scale hyperparameters with width ("how should my initialization and learning rate change when my width doubles?"), which is coarser than a prescription for setting hyperparameters at any particular width ("how should I set my initialization and learning rate at width 1024?").

11 Conclusion

Leveraging the discovery of a feature learning neural network infinite-width limit, we hypothesized and verified that the HP landscape across NNs of different width is reasonably stable if parametrized according to Maximal Update Parametrization (μP). We further empirically showed that it's possible to transfer across depth, batch size, sequence length, and training time, with a few caveats. This allowed us to indirectly tune a very large network by tuning its smaller counterparts and transferring the HPs to the full model. Our results raise an interesting new theoretical question of how *useful HP transfer* is possible in neural networks in the first place.

Venues of Improvement Nevertheless, our method has plenty of room to improve. For example, initialization does not transfer well across depth, and depth transfer generally still does not work for post-layernorm Transformers. This begs the question whether a more principled parametrization in depth could solve these problems. Additionally, Fig. 4 shows that the optimal HP still shifts slightly for smaller models. Perhaps by considering finite-width corrections to μP one can fix this shift. Finally, it will be interesting to study if there's a way to transfer regularization HPs as a function of both the model size and data size, especially in the context of finetuning of pretrained models.

Acknowledgements In alphabetical order, we thank Arthur Jacot, Arturs Backurs, Colin Raffel, Denny Wu, Di He, Huishuai Zhang, Ilya Sutskever, James Martens, Janardhan Kulkarni, Jascha Sohl-Dickstein, Jeremy Bernstein, Lenaic Chizat, Luke Metz, Mark Chen, Michael Santacroce, Muhammad ElNokrashy, Pengchuan Zhang, Sam Schoenholz, Sanjeev Arora, Taco Cohen, Yiping Lu, Yisong Yue, and Yoshua Bengio for discussion and help during our research.

References

[1] NVIDIA/DeepLearningExamples, apache v2 license. URL https://github.com/NVIDIA/DeepLearningExamples.

[2] Davidnet, mit license, 2019. URL https://github.com/davidcpage/cifar10-fast.

[3] Autonomio talos, mit license, 2019. URL http://github.com/autonomio/talos.

[4] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework, 2019.

[5] Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W. Cottrell, and Julian McAuley. ReZero is All You Need: Fast Convergence at Large Depth. *arXiv:2003.04887 [cs, stat]*, June 2020. URL http://arxiv.org/abs/2003.04887.

[6] Jeremy Bernstein, Arash Vahdat, Yisong Yue, and Ming-Yu Liu. On the distance between two neural networks and the stability of learning. *arXiv:2002.03432 [cs, math, stat]*, January 2021. URL http://arxiv.org/abs/2002.03432.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[8] Simon Carbonnelle and Christophe De Vleeschouwer. Layer rotation: a surprisingly powerful indicator of generalization in deep networks? *arXiv:1806.01603 [cs, stat]*, July 2019. URL http://arxiv.org/abs/1806.01603.

[9] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling, 2014.

[10] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]*, May 2019. URL http://arxiv.org/abs/1810.04805.

[12] Xiaohan Ding, Chunlong Xia, Xiangyu Zhang, Xiaojie Chu, Jungong Han, and Guiguang Ding. RepMLP: Re-parameterizing Convolutions into Fully-connected Layers for Image Recognition. *arXiv:2105.01883 [cs]*, August 2021. URL http://arxiv.org/abs/2105.01883.

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, May 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

[14] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *arXiv:1705.08741 [cs, stat]*, May 2017. URL http://arxiv.org/abs/1705.08741.

[15] Samuel Horváth, Aaron Klein, Peter Richtárik, and Cédric Archambeau. Hyperparameter transfer learning with adaptive complexity. *CoRR*, abs/2102.12810, 2021. URL https://arxiv.org/abs/2102.12810.

[16] Xiao Shi Huang and Felipe Pérez. Improving Transformer Optimization Through Better Initialization. page 9.

[17] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural Tangent Kernel: Convergence and Generalization in Neural Networks. *arXiv:1806.07572 [cs, math, stat]*, June 2018. URL http://arxiv.org/abs/1806.07572.

[18] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization, 2015.

[19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv:2001.08361 [cs, stat]*, January 2020. URL http://arxiv.org/abs/2001.08361.

[20] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning, 2020.

[22] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *JMLR 18*, page 52.

[23] Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay Attention to MLPs. *arXiv:2105.08050 [cs]*, June 2021. URL http://arxiv.org/abs/2105.08050.

[24] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.463. URL https://www.aclweb.org/anthology/2020.emnlp-main.463.

[25] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the Difficulty of Training Transformers. *arXiv:2004.08249 [cs, stat]*, September 2020. URL http://arxiv.org/abs/2004.08249.

[26] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4487–4496, Florence, Italy, July 2019. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P19-1441.

[27] Yang Liu, Jeremy Bernstein, Markus Meister, and Yisong Yue. Learning by Turning: Neural Architecture Aware Optimisation. *arXiv:2102.07227 [cs]*, September 2021. URL http://arxiv.org/abs/2102.07227.

[28] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An Empirical Model of Large-Batch Training. *arXiv:1812.06162 [cs, stat]*, December 2018. URL http://arxiv.org/abs/1812.06162.

[29] Luke Melas-Kyriazi. Do You Even Need Attention? A Stack of Feed-Forward Layers Does Surprisingly Well on ImageNet. *arXiv:2105.02723 [cs]*, May 2021. URL http://arxiv.org/abs/2105.02723.

[30] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[31] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling, mit license. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[32] Daniel S. Park, Jascha Sohl-Dickstein, Quoc V. Le, and Samuel L. Smith. The Effect of Network Width on Stochastic Gradient Descent and Generalization: an Empirical Study. May 2019. URL https://arxiv.org/abs/1905.03776v1.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, bsd-style license. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dÁlché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[34] Valerio Perrone, Rodolphe Jenatton, Matthias W Seeger, and Cedric Archambeau. Scalable Hyperparameter Transfer Learning. *NeurIPS 2018*, page 11.

[35] Martin Popel and Ondřej Bojar. Training Tips for the Transformer Model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70, April 2018. ISSN 1804-0462. doi: 10.2478/pralin-2018-0002. URL http://content.sciendo.com/view/journals/pralin/110/1/article-p43.xml.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683 [cs, stat]*, July 2020. URL http://arxiv.org/abs/1910.10683.

[37] Samuel S. Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep Information Propagation. *arXiv:1611.01232 [cs, stat]*, November 2016. URL http://arxiv.org/abs/1611.01232.

[38] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *arXiv:1811.03600 [cs, stat]*, November 2018. URL http://arxiv.org/abs/1811.03600.

[39] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. April 2018. URL https://arxiv.org/abs/1804.04235v1.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

[41] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv:1711.00489 [cs, stat]*, November 2017. URL http://arxiv.org/abs/1711.00489.

[42] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

[43] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks, 2015.

[44] Danny Stoll, Jörg K.H. Franke, Diane Wagner, Simon Selg, and Frank Hutter. Hyperparameter transfer across developer adjustments, 2021. URL https://openreview.net/forum?id=WPO0vDYLXem.

[45] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. MLP-Mixer: An all-MLP Architecture for Vision. *arXiv:2105.01601 [cs]*, June 2021. URL http://arxiv.org/abs/2105.01601.

[46] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Gautier Izacard, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. ResMLP: Feedforward networks for image classification with data-efficient training. *arXiv:2105.03404 [cs]*, June 2021. URL http://arxiv.org/abs/2105.03404.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[48] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *EMNLP 2018*, page 353, 2018.

[49] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018. URL http://aclweb.org/anthology/N18-1101.

[50] Greg Yang. Tensor Programs I: Wide Feedforward or Recurrent Neural Networks of Any Architecture are Gaussian Processes. *arXiv:1910.12478 [cond-mat, physics:math-ph]*, December 2019. URL http://arxiv.org/abs/1910.12478.

[51] Greg Yang. Scaling Limits of Wide Neural Networks with Weight Sharing: Gaussian Process Behavior, Gradient Independence, and Neural Tangent Kernel Derivation. *arXiv:1902.04760 [cond-mat, physics:math-ph, stat]*, February 2019. URL http://arxiv.org/abs/1902.04760.

[52] Greg Yang. Tensor Programs II: Neural Tangent Kernel for Any Architecture. *arXiv:2006.14548 [cond-mat, stat]*, August 2020. URL http://arxiv.org/abs/2006.14548.

[53] Greg Yang. Tensor Programs III: Neural Matrix Laws. *arXiv:2009.10685 [cs, math]*, September 2020. URL http://arxiv.org/abs/2009.10685.

[54] Greg Yang and Edward J. Hu. Feature learning in infinite-width neural networks. *arXiv*, 2020.

[55] Greg Yang and Etai Littwin. Tensor Programs IIb: Architectural Universality of Neural Tangent Kernel Training Dynamics. *arXiv:2105.03703 [cs, math]*, May 2021. URL http://arxiv.org/abs/2105.03703.

[56] Greg Yang and Sam S. Schoenholz. Deep Mean Field Theory: Layerwise Variance and Width Variation as Methods to Control Gradient Explosion. February 2018. URL https://openreview.net/forum?id=rJGY8GbR-.

[57] Greg Yang and Samuel S. Schoenholz. Mean Field Residual Networks: On the Edge of Chaos. *arXiv:1712.08969 [cond-mat, physics:nlin]*, December 2017. URL http://arxiv.org/abs/1712.08969.

[58] Greg Yang, Michael Santacroce, and Edward J Hu. Efficient computation of deep nonlinear infinite-width neural networks that learn features. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=tUMr0Iox8XW.

[59] Dani Yogatama and Gideon Mann. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In *Artificial Intelligence and Statistics*, pages 1077–1085. PMLR, April 2014. URL http://proceedings.mlr.press/v33/yogatama14.html.

[60] Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks. *arXiv:1708.03888 [cs]*, September 2017. URL http://arxiv.org/abs/1708.03888.

[61] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. *arXiv:1904.00962 [cs, stat]*, January 2020. URL http://arxiv.org/abs/1904.00962.

[62] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2017.

[63] Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Residual Learning Without Normalization via Better Initialization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gsz30cKX.

Contents

List of Figures

List of Tables

A Parametrization Terminologies

This section seeks to make formal and clarify some of the notions regarding parametrization discussed informally in the main text.

Definition A.1 (Multiplier and Parameter Multiplier)**.** In a neural network, one may insert a "multiply by c" operation anywhere, where c is a non-learnable scalar hyperparameter. If $c = 1$, then this operation is a no-op. This c is called a *multiplier*.

Relatedly, for any parameter tensor W in a neural network, we may replace W with cW for some non-learnable scalar hyperparameter c. When $c = 1$, we recover the original formulation. This c is referred to as a *parameter multiplier*.

For example, in the attention logit calculation $\langle k, q\rangle/\sqrt{d_{head}}$ where $q = Wx$, the $1/\sqrt{d_{head}}$ factor is a multiplier. It may also be thought of as the parameter multiplier of W if we rewrite the attention logit as $\langle k, (W/\sqrt{d_{head}})x\rangle$.

Note parameter multipliers cannot be absorbed into the initialization in general, since they affect backpropagation. Nevertheless, after training is done, parameter multipliers can always be absorbed into the weight.

Definition A.2 (Parametrization)**.** In this work, a *parametrization* is a rule for how to *change* hyperparameters when the widths of a neural network *change*, but note that it does not necessarily prescribes how to set the hyperparameters for any specific width. In particular, for any neural network, an *abc-parametrization* is a rule for how to scale a) the parameter multiplier, b) the initialization, and c) the learning rate individually for each parameter tensor as the widths of the network change, as well as any other multiplier in the network; all other hyperparameters are kept fixed with width.

For example, SP and μP are both abc-parametrizations. Again, we note that, in this sense, a parametrization does not prescribe, for example, that the initialization variance be $1/\text{fan\_in}$, but rather that it be halved when fan_in doubles.

Definition A.3 (Zero-Shot Hyperparameter Transfer)**.** In this work, we say a parametrization admits *zero-shot transfer of a set of hyperparameters $\mathcal{H}$ w.r.t. a metric $\mathcal{L}$* if the optimal combination of values of $\mathcal{H}$ w.r.t. $\mathcal{L}$ converges as width goes to infinity, i.e. it stays approximately optimal w.r.t. $\mathcal{L}$ under this parametrization as width increases.

Throughout this paper, we take $\mathcal{L}$ to be the training loss, but because regularization is not the bottleneck in our experiments (especially large scale pretraining with BERT and GPT-3), we nevertheless see high quality test performance in all of our results. We also remark that empirically, using training loss as the metric can be more robust to random seed compared to validation loss and especially BLEU score. See Table 1(left) for $\mathcal{H}$. By our arguments in Appendix J.3 and our empirical results, μP is the unique abc-parametrization admitting zero-shot transfer for such $\mathcal{H}$ and $\mathcal{L}$ in this sense.

More generally, one may define a *K-shot transfer algorithm of a set of hyperparameters $\mathcal{H}$ w.r.t. a metric $\mathcal{L}$* as one that 1) takes width values n and n' and an approximately optimal combination of values of $\mathcal{H}$ w.r.t. $\mathcal{L}$ at a width n and 2) returns an approximately optimal combination of values of $\mathcal{H}$ w.r.t. $\mathcal{L}$ at width n', given 3) a budget of K evaluations of candidate hyperparameter combinations on models of width n'. However, we will have no use for this definition in this paper.

B Further Explanations of the μP Tables

In addition to Table 3, we provide Table 8 as an equivalent μP formulation that is easier to implement, as well as Table 9 for those more familiar with the original μP formulation in [54]. Below, we provide some commentary on corner cases not well specified by the tables. Ultimately, by understanding Appendix J, one can derive μP for any architecture, new or old.

Matrix-Like, Vector-Like, Scalar-Like Parameters We can classify any dimension in a neural network as "infinite" if it scales with width, or "finite" otherwise. For example, in a Transformer, $d_{model}, d_{ffn}, d_{head}, n_{head}$ are all infinite, but vocab size and context size are finite. Then we can categorize parameter tensors by how many infinite dimensions they have. If there are two such dimensions, then we say the parameter is *matrix-like*; if there is only one, then we say it is *vector-like*; if there is none, we say it is *scalar-like*. Then in Tables 3, 8 and 9, "input weights & all biases" and "output weights" are all vector-like parameters, while hidden weights are matrix-like parameters. An

Table 8: **Alternative (Equivalent) μP Formulation for Easier Implementation.** Same format as in Table 3. In contrast to the formulation in Table 3, here all "vector-like" parameters (i.e. those that have only one dimension tending to infinity), including input and output weights and biases, have the same width scaling for initialization variance and SGD/Adam LR (note the $1/\text{fan\_in}$ for input weight/bias init. var. is $\Theta(1)$ in width). This has two benefits in practice: 1) implementation is unified and simplified for all "vector-like" parameters; 2) input and output weights can now be tied, in contrast to Table 3, which is a common design feature of Transformer models. Note that in this table, for biases, the fan_in is 1 (compare to PyTorch `nn.Linear` default initialization of biases, where fan_in refers to fan_in of the layer.) This table can be derived from Table 3 via Lemma J.1. See Appendix B for further explanations.

	Input weights & all biases	Output weights	Hidden weights
Init. Var.	$1/\text{fan\_in}$	1 $(1/\text{fan\_in})$	$1/\text{fan\_in}$
Multiplier	1	$1/\text{fan\_in}$ (1)	1
SGD LR	fan_out (1)	fan_in (1)	1
Adam LR	1	1	$1/\text{fan\_in}$ (1)

Table 9: **μP Formulation in the Style of [54].** This table can be derived from Table 3 via Lemma J.1.

	Input weights & all biases	Output weights	Hidden weights
Init. Var.	$1/\text{fan\_out}$ $(1/\text{fan\_in})$	$1/\text{fan\_in}$	$1/\text{fan\_in}$
Multiplier	$\sqrt{\text{fan\_out}}$ (1)	$1/\sqrt{\text{fan\_in}}$ (1)	1
SGD LR	1	1	1
Adam LR	$1/\sqrt{\text{fan\_out}}$ (1)	$1/\sqrt{\text{fan\_in}}$ (1)	$1/\text{fan\_in}$ (1)

advantage of Table 8 is that it gives a uniform scaling rule of initialization and learning rate for all vector-like parameters. The multiplier rule in Table 8 can be more interpreted more generally as the following: a multiplier of order $1/\text{fan\_in}$ should accompany any weight that maps an infinite dimension to a finite one. This interpretation then nicely covers both the output logits and the attention logits (i.e. $1/d$ attention).

Scalar-like parameters are not as common as matrix-like and vector-like ones, but we will mention a few examples in Appendix B.2. The scaling rule for their initialization, learning rate (for both SGD and Adam), and multiplier is very simple: hold them constant with width.

Initialization Mean We did not specify the initialization mean in the tables, since most commonly the mean is just set to 0, but it can be nonzero for vector-like parameters (e.g., layernorm weights) and scalar-like parameters but must be 0 for matrix-like parameters.

Zero Initialization Variance The initialization scaling rules in our tables can all be trivially satisfied if the initialization variance is set to 0. This can be useful in some settings (e.g., Appendix D.2) but detrimental in other settings (e.g., hidden weights).

What Are Considered Input Weights? Output Weights? Here, input weights very specifically refer to weights that map from an infinite dimension to a finite dimension. As a counterexample, in some architectures, the first layer can actually map from a finite dimension to another finite dimension, e.g., a PCA layer. Then this is not an "input weight"; if the next layer maps into an infinite dimension, then that's the input weight. A similar, symmetric discussion applies to output weights.

What Counts As a "Model"? Does the MLP in a Transformer Count As a "Model"? For our tables, a model is specifically a function that maps a finite dimension to another finite dimension, consistent with the discussion above. For example, for an image model on CIFAR10, it maps from $3 \times 32 \times 32 = 3072$ dimensions to 10 dimensions, and these numbers are fixed regardless of the width of the model. Likewise, for an autoregressive Transformer model, the input and output dimension are both the vocab size, which is independent of the width. In contrast, an MLP inside a Transformer is not a "model" in this sense because its input and output dimension are both equal to the width of the Transformer.

B.1 Walkthrough of μP Implementation in a Transformer

To ground the abstract description in Tables 3, 8 and 9, we walk through the parameters of a typical Transformer and discuss concretely how to parametrize each.

We assume that the user wants to replicate SP when the model widths are equal to some base widths, for example, when $d_{model} = d_{model,0} = 128, d_{ffn} = d_{ffn,0} = 512$, etc, as in the MLP example in Section 4. For this purpose, it's useful to define $\tilde{d}_{model} = d_{model}/d_{model,0}$, $\tilde{d}_{ffn} = d_{ffn}/d_{ffn,0}$, and so on. One can always take $d_{model,0} = d_{ffn,0} = \cdots = 1$ for a "pure" μP.

Below, we introduce hyperparameters $\sigma_\bullet, \eta_\bullet$ for each parameter tensor, as well as a few multipliers $\alpha_\bullet$. One may always tie $\sigma_\bullet$ (resp. $\eta_\bullet$) across all parameter tensors, but in our experiments, we found it beneficial to at least distinguish the input and output layer initialization and learning rates.

Input Word Embeddings The input word embedding matrix $W^{wordemb}$ has size $d_{model} \times vocabsize$, where $vocabsize$ is the fan-in and d_{model} is the fan-out. Follow the "input weight & all biases" column in Tables 3, 8 and 9. For example, for Tables 3 and 8,

$$W^{wordemb} \sim \mathcal{N}(0, \sigma^2_{wordemb}), \quad \text{with Adam LR } \eta_{wordemb}$$

Note here, because fan-in ($vocabsize$) here is independent of width (d_{model}), the "$1/\text{fan\_in}$" for the initialization variance in these tables is equivalent to "1", i.e. the initialization variance can be anything fixed with width. In this case of the word embedding, setting the variance to 1, for example, is more natural than setting the variance to $1/\text{fan\_in}$, because the embedding is one-hot ($1/\text{fan\_in}$ would be more natural for image inputs).

Positional Embeddings The (absolute or relative) positional embedding matrix W^{posemb} has size $d_{model} \times contextsize$, where $contextsize$ is the fan-in and d_{model} is the fan-out. With the same discussion as above for input word embeddings, follow the "input weight & all biases" column in Tables 3, 8 and 9. For example, for Tables 3 and 8,

$$W^{posemb} \sim \mathcal{N}(0, \sigma^2_{posemb}), \quad \text{with Adam LR } \eta_{posemb}$$

Layernorm Weights and Biases Layernorm weights w^{LN} and biases b^{LN} both have shape d_{model} and can be thought of "input weights" to the scalar input of 1. Hence one should follow the "input weight & all biases" column in Tables 3, 8 and 9. In particular, the usual initialization of layernorm weights as all 1s and biases as all 0s suffice (where the initialization variance is 0). For example, for Tables 3 and 8,

$$w^{LN} \leftarrow 1, \quad \text{with Adam LR } \eta_{LNw}, \qquad \text{and} \qquad b^{LN} \leftarrow 0, \quad \text{with Adam LR } \eta_{LNb}$$

Self-Attention There are 4 matrices, $W^q, W^k \in \mathbb{R}^{(d_k n_{head}) \times d_{model}}$, $W^v \in \mathbb{R}^{(d_v n_{head}) \times d_{model}}$, and $W^o \in \mathbb{R}^{d_{model} \times (d_v n_{head})}$ (where the shapes are $\mathbb{R}^{\text{fan\_out} \times \text{fan\_in}}$). Since d_{model}, $(d_k n_{head})$, and $(d_v n_{head})$ all scale with width (where the latter two are commonly just set to d_{model}), all 4 matrices should be parametrized according to the "hidden weights" column in Tables 3, 8 and 9. For example, for Tables 3 and 8,

$$\begin{aligned}
W^q &\sim \mathcal{N}(0, \sigma_q^2/d_{model}), && \text{with Adam LR } \eta_q/\tilde{d}_{model} \\
W^k &\sim \mathcal{N}(0, \sigma_k^2/d_{model}), && \text{with Adam LR } \eta_k/\tilde{d}_{model} \\
W^v &\sim \mathcal{N}(0, \sigma_v^2/d_{model}), && \text{with Adam LR } \eta_v/\tilde{d}_{model} \\
W^o &\sim \mathcal{N}(0, \sigma_o^2/(d_v n_{head})), && \text{with Adam LR } \eta_o/(\tilde{d}_v \tilde{n}_{head}).
\end{aligned}$$

Attention Logit Scaling We use $1/d$ instead of $1/\sqrt{d}$ attention. To be compatible with $1/\sqrt{d}$ attention when at a particular base $d_{head} = d_{head,0}$, we set

$$AttnLogit = \alpha_{attn} \frac{\sqrt{d_{head,0}}}{d_{head}} q^\top k,$$

where α_{attn} is a tunable multiplier.

MLP There are 2 matrices, $W^1 \in \mathbb{R}^{d_{ffn} \times d_{model}}, W^2 \in \mathbb{R}^{d_{model} \times d_{ffn}}$ (where the shapes are $\mathbb{R}^{\text{fan\_out} \times \text{fan\_in}}$), where d_{ffn} is commonly set to $4d_{model}$. Since both d_{model}, d_{ffn} scale with width, both matrices are considered "hidden weights." For example, for Tables 3 and 8,

$$W^1 \sim \mathcal{N}(0, \sigma_q^2/d_{model}), \qquad \text{with Adam LR } \eta_q/\tilde{d}_{model}$$
$$W^2 \sim \mathcal{N}(0, \sigma_k^2/d_{ffn}), \qquad \text{with Adam LR } \eta_k/\tilde{d}_{ffn}$$

Word Unembeddings Symmetric to the discussion on input word embeddings, the output word unembeddings should be parametrized according to the "output weights" column of Tables 3, 8 and 9. Often, the unembeddings are tied with the embeddings, and Tables 8 and 9 allow for this as their initialization schemes are symmetric between input and output weights.

For example, for Table 3, we'd set

$$W^{unemb} \sim \mathcal{N}(0, \sigma_{unemb}^2/(d_{model}\tilde{d}_{model})), \quad \text{with Adam LR } \eta_{unemb}/\tilde{d}_{model}.$$

For Table 8, we would instead have

$$W^{unemb} \sim \mathcal{N}(0, \sigma_{unemb}^2/d_{model,0}), \quad \text{with Adam LR } \eta_{unemb},$$

(note $d_{model,0}$ here is the base width and therefore is a constant) and the output is computed as

$$logits = \frac{\alpha_{output}}{\tilde{d}_{model}} W^{unemb} z$$

where z is the final layer embedding of a token, and α_{output} is a tunable multiplier.

B.2 Other Parameters

Learnable scalar multipliers For learnable scalar multipliers (e.g., softmax inverse temperature), one can initialize them to 1 and use a constant (in width) learning rate for both SGD and Adam. This is compatible with Tables 3, 8 and 9.

Positional Bias Some Transformers use positional bias (of size $contextsize \times contextsize$, which are added to the attention logits). They are considered "scalar-like" in that it has no width dimension. One can initialize them to 0 and use a constant (in width) learning rate for both SGD and Adam. This is compatible with Tables 3, 8 and 9.

Spatial MLPs Recent works [12, 23, 29, 45, 46] on MLP-only architectures in NLP and CV replace the self-attention layer in Transformers with MLPs across tokens or spatial locations. In our language here, such MLPs have finite input and output dimensions (the context size) and infinite hidden dimensions, so their input, output, and hidden weights should be parametrized via the corresponding columns in Tables 3, 8 and 9.

B.3 Optimizer Variants and Hyperparameters

AdamW Exactly the same as Adam in all of our tables, with the added benefit that weight decay is automatically scaled correctly in AdamW (but is incompatible with μP Adam). For this reason, we recommend using AdamW when weight decay is desired (which is consistent with current standard practice).

Frobenius Normalization LARS [60], Adafactor [39], Lamb [61], Layca [8], Fromage [6], Nero [27] all involve a normalization step in which the update g (which may be obtained from SGD, Adam, or other optimzers) is normalized to have Frobenius norm equal to that of the parameter w: $g \leftarrow \frac{\|w\|_F}{\|g\|_F} g$. They can be made compatible with μP in Table 8 by scaling their learning rate for hidden weights like $1/\sqrt{\text{fan\_in}}$ (for Table 3, the output weight learning rate should be likewise scaled). The intuitive reasoning (which can be formalized straightforwardly using Tensor Programs) is as follows.

This normalization implicitly encodes a width scaling: If one initializes a weight matrix with variance $1/\text{fan\_in}$, then an $n \times n$ matrix (e.g., a hidden weight matrix) has Frobenius norm $\sqrt{n}$ at initialization. Thus, in the first step and, by induction, in any step t, the normalized update to this $n \times n$ weight also

has Frobenius norm $\Theta(\sqrt{n})$ (for any fixed t, as $n \to \infty$). Heuristically, this means each entry of g is approximately of size $\Theta(1/\sqrt{n})$. But, by the derivation of Appendix J, we want $\Theta(1/n)$ and this is $\Theta(\sqrt{n})$ too large! Thus, in wide enough networks, one should see a network blowup after one update, like demonstrated in Fig. 5.

However, note that the $\Theta(1/\sqrt{n})$ coordinate size induced by the normalization here is closer to the right size $\Theta(1/n)$ than Adam, whose update have coordinate size $\Theta(1)$. This may partially explain the apparent benefit of these optimizers. In particular, this may explain the observation that T5 [36], using Adafactor, was able to train its entire range of models from 220 million to 11 billion parameters with a fixed set of hyperparameters, while GPT-3 [7], using Adam, needed to decrease its learning rate with model size.

RAdam RAdam [24] is a variant of Adam that uses SGD with momentum in an initial stage with learning rate warmup, followed by a second stage of Adam with a particular setting of learning rate with time. Thus, one can adapt RAdam to μP by individually scaling the learning rates of the initial SGD stage and the final Adam stage according to Table 3, Table 8, or Table 9.

Adagrad and RMSProp Exactly the same as Adam in all of our tables.

ϵ in Adam and Its Variants All of our derivations here assume ϵ is negligible in Adam. If it is set to a non-negligible number, then it needs to be scaled, for all parameters, like $1/\text{fan\_in}^2$ if it is added before the square root, or like $1/\text{fan\_in}$ if it is added after the square root.

Gradient Clipping Gradient (ℓ_2-norm-wise) clipping is compatible with Table 3 (as well as Tables 8 and 9), for either SGD or Adam, if the clip value is held constant with respect to width.

Weight Decay Weight decay should be scaled independently of width in SGD and AdamW, for all of our tables. However, note it's not compatible with μP Adam.

Momentum Momentum should be scaled independently of width for all of our tables.

C Parametrization Matters: A Primer for Multiple Hyperparameters

Here we give more intuition why we need to reparametrize *all* hyperparameters. In practice, neural networks have multitudes of hyperparameters all interacting together. In our example of Section 2, hyperparameter optimization would be akin to minimizing the function[19]

$$F_n(c^1, \dots, c^k) \overset{\text{def}}{=} \mathop{\mathbb{E}}_{x_1,\dots,x_n} f((c^1 + \cdots + c^k)(x_1 + \cdots + x_n)).$$

where $x_1, \dots, x_n$ are as in Eq. (1) and $c^1, \dots, c^k$ are analogous to k hyperparameters. For the same reasoning in Section 2, the *correct parametrization* is in $(\alpha^1, \dots, \alpha^k)$ where $\alpha^i = c^i\sqrt{n}$.

While this is straightforward, in practice, researchers often fix some hyperparameters (e.g., they tune only learning rate but neglects to scale parameter multipliers or initialization correctly). For example, if we only partially reparametrize and optimize in α^1 while fixing $c^2, \dots, c^k$, then the optimal α^1 is $(\alpha^1)^* = \alpha^* - (c^1 + \ldots + c^k)\sqrt{n}$ where α^* is the optimal α for Eq. (1). Thus, as $n \to \infty$, $(\alpha^1)^*$ still blows up even though we parametrized α^1 correctly. More generally, the incorrect parametrization of some hyperparameters forces other hyperparameters to increasingly compensate for it as width grows, distorting their optima, even if the latter are correctly parametrized.

D Practical Considerations

In this section, we outline several useful tips and tricks that can improve the quality of hyperparameter transfer in practice.

[19] Here, for simplicity of the example, we model the interaction between "hyperparameters" $c^1, \dots, c^k$ as additive, but in real neural networks such interactions are usually much more complicated.

D.1 Verifying μP Implementation via *Coordinate Checking*

Even though μP is neatly encapsulated by Table 3, implementing it correctly can in practice be error-prone, just like how implementing autograd by hand can be error-prone even though the math behind is just chain-rule. In the case of autograd, gradient checking is a simple way of verifying implementation correctness; similarly, we propose *coordinate checking* to verify the correctness of μP implementation: Exemplified by Fig. 5, one calculates the average coordinate size of every (pre)activation vector in the network over a few steps of training, as width is varied over a large range. An incorrect implementation will see some activation vector blow up or shrink to zero with width (like in the top row of Fig. 5). In the `mup` package we release with this paper, we include an easy-to-use method for coordinate checking.

D.2 Zero Initialization for Output Layers and Query Layers in Attention

We find that the optimal hyperparameters of small and large width models match more closely when we initialize output layers at 0 (i.e. with variance $\sigma^2/\text{fan\_in}$ where $\sigma = 0$ instead of positive σ). This is because the neural network in μP is approximately a Gaussian process (GP) at initialization with variance on the order $\Theta(\sigma^2/width)$ (contrast this with SP networks, which approximates a GP with $\Theta(\sigma^2)$ variance) [50, 54]. Of course, when width is large, this variance vanishes, but this can be far from so in the small proxy model. This discrepancy in the initial GP can cause the training trajectory of the proxy model to be very different from the trajectory of the large target model, causing a mismatch in the optimal hyperparameters. By initializing the output layer at 0, we remove this mismatch in the initial GP. Empirically we do not find this modification to be detrimental to performance.

A similar consideration applies to the query layer in self-attention, and more generally, any layer that goes from an "infinite" dimension (i.e. width) to a "finite" dimension (e.g. output dimension or sequence length).

D.3 Activation Functions

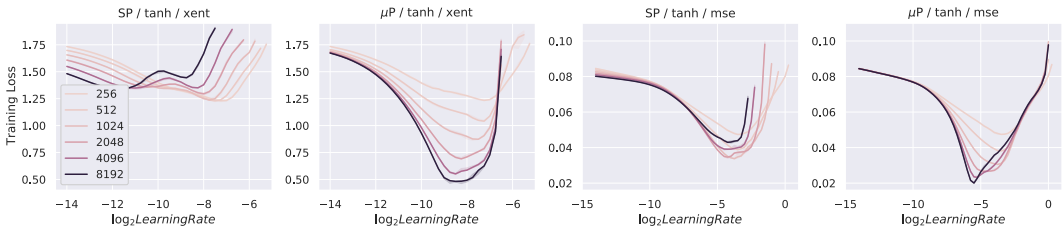


Figure 9: **Squashing activation functions reduce transfer quality.** MLP of different hidden sizes with `tanh` activation trained for 20 epoch on CIFAR-10 using SGD. Left uses cross-entropy as loss function; right uses mean squared error; columns alternate between standard parametrization (SP) and maximal update parametrization (μP). Compared to `ReLU`, `tanh` exhibits slower convergence for μP, yet it still outperforms SP when width is increased

When the network is narrow, its approximation to the infinite-width behavior becomes crude, which is manifested as large fluctuations in preactivation coordinates. When using a squashing activation functions like `softmax` or `tanh`, this causes narrower networks to saturate the activation more than wider ones, which results in a systematic bias toward small gradients and therefore distorting the hyperparameter landscape. This can be seen in Fig. 9, where we use `tanh` as the network activation function.

Therefore, we recommend replacing non-essential squashing activation functions with `ReLU`, whose derivative depends only on the sign of the pre-activation. A similar reasoning can be applied to superlinear activation functions, where the distribution of activation values can have heavy tails, leading to slow convergence to the infinite-width limit. However, such activations are rarely used in practice.

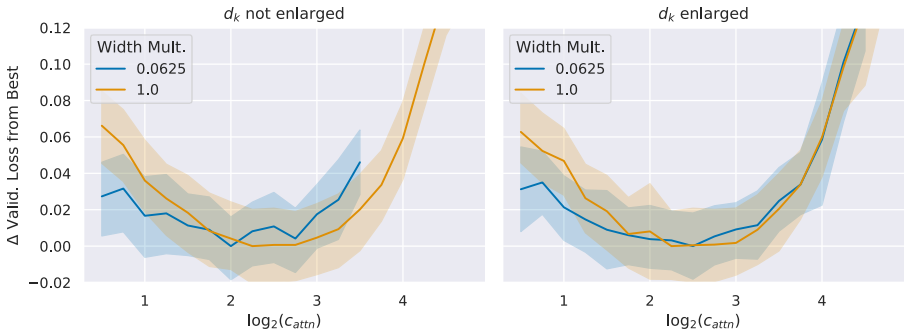


Figure 10: **Enlarging d_k makes μTransfer more precise.** Here we plot all curves *after subtracting their minima* for easier visual comparison. Transformer on IWSLT 14 similar to the setup in Appendix F.1 where the $d_{model} = 512$ for a width multiplier of 1, $n_{head} = 4$, and $d_q = d_k$. **(Left)** We leave $d_q = d_k = {}^{d_{model}}/_{n_{head}}$, so $d_k = 8$ for width-multiplier 0.0625. The optimum for the attention logit multiplier c_{attn} is noisy and does not accurately transfer across width. **(Right)** We enlarge $d_q = d_k$ to a minimum of 128. The HP landscape is much smoother than in (Left), and the optima align between narrow and wide models.

D.4 Enlarge d_k

We find that small $d_{head} = d_k$ can lead to a highly noisy HP landscape, as shown in Fig. 10. This can significiantly decrease the quality of random HP search on the small proxy model. To solve this, we find it useful to decouple d_k from d_{model} (so that $d_{model} \neq d_k \cdot n_{head}$) and maintain a relatively large d_k even as d_{model} is shrunk in the proxy model. For example, pegging $d_k = 32$ is generally effective. Training or inference speed are not usually affected much by the larger d_k because of CUDA optimizations. By Appendix E.2, this decoupling of d_k from d_{model} is theoretically justified, and as shown in Fig. 10, it significantly denoises the HP landscape.

D.5 Non-Gaussian vs Gaussian Initialization

We find non-Gaussian (e.g. uniform) initialization can sometimes cause wider models to perform worse than narrower models, whereas we do not find this behavior for Gaussian initialization. This is consistent with theory, since in the large width limit, one should expect non-Gaussian initialization to behave like Gaussian initializations anyway (essentially due to Central Limit Theorem, or more precisely, universality), but the non-Gaussianity slows down the convergence to this limit.

D.6 Using a Larger Sequence Length

For Transformers, we empirically find that we can better transfer initialization standard deviation from a narrower model (to a wide model) if we use a larger sequence length. It is not clear why this is the case. We leave an explanation to future work.

D.7 Tuning Per-Layer Hyperparameters

The techniques in this paper allow the transfer across width of (learning rate, initialization, multipliers) simultaneously for all parameter tensors. Thus, to get the best results, one should ideally tune all such hyperparameters. In practice, we find that just tuning the global learning rate and initialization, along with input, output, and attention multipliers, yield good results.

E Which Hyperparameters Can Be Transferred? (Continued)

E.1 Further Discussions on Hyperparameter Categories

Below, we discuss the reasoning behind each kind, which are supported by our empirical evidence collected in Fig. 4 on Transformers as well as those in Appendix G.1 on ResNet.

Transferable Hyperparameters In Table 2, we summarize which HPs can be transferred across training scale. The transfer across *width*, as explained in Section 2, is theoretically justified, while we present the transfer across the other dimensions as empirical results.

These cover most of the well-known and important HPs when the need for regularization is not paramount, e.g., during large scale language model pretraining. Parameter Multipliers are not well-known HPs, yet we include them here as they serve a bridge between SP and μP and can impact model performance in practice. Concretely, any SP and μP neural networks of the same width can have their Parameter Multipliers tuned so that their training dynamics become identical.

Hyperparameters That Don't Transfer Well Not all HPs transfer well even if we use μP. In particular, those whose primary function is to regularize training to mitigate "overfitting" tend not to transfer well. Intuitively, regularization needs to be applied more heavily in larger models and when data is scarce, but μP does not know the data size so cannot adjust the regularization accordingly.

To the best of our knowledge, there is no strict separation between HPs that regularize and those that don't. However, conventional wisdom tells us that there exists a spectrum of how much regularizing effect a HP has. For example, dropout probability and weight decay are among those whose primary function is to regularize, whereas batch size and learning rate might regularize training in some cases but affect the dynamics more so in other ways. Our empirical exploration tells us that the former do not transfer well, while the latter do. Our subsequent discussion will focus on the latter; we leave to future works the expansion to the former.

Hyperparameters Transfered *Across* We have left out a category of HPs that defines the training *scale*, or in practical terms, training cost. This includes 1) those that define how many operations a model's forward/backward pass takes, such as the model's width, depth, and in the case of language modeling, sequence length; and 2) those that define how many such passes are performed, such as batch size and number of training steps.

As recent works have shown [7, 19], improvements along any of these *scale* dimensions lead to apparently sustainable gain in performance; as a result, we are primarily interested in transferring other HPs *across* these dimensions that define scale, rather than finding the optimal scale.[20] This category of HPs is particularly crucial as one can speedup training by downsizing in one or multiple such dimensions. Indeed, it's very common for practitioners to implicitly transfer HPs across the number of training samples by tuning on only a subset of the full training data.

Our insights from the infinite-width limit inspired us to explore HP tranfer across *width*, which does not work under SP as we have shown earlier. Building upon our success with width, which is well explained theoretically, we hope to push the limit of compute-saving by investigating the other dimensions empirically. To the best of our knowledge, the transferability of optimal HPs across depth, batch size, sequence length, and training time has not been rigorously investigated previously, with the main exception of the literature on (learning rate, batch size) scaling [38, 41] where our transferability result of learning rate across batch size recapitulates [28].[21] See Section 10.3 on how our results relate to prior works. We will primarily focus on the Transformer architecture in the main text with evidence for ResNet in Appendix G.1.

E.2 On the Definitions of Width

Our theory allows more general notions of width. This is especially relevant in Transformers, where $d_{model}, d_{head} = d_k, d_v, n_{head}, d_{ffn}$ (see Fig. 11) can all be construed as measures of width. We briefly discuss these here, with more theoretical justification in Appendix J.2.1 and empirical validation below.

Varying Width Ratio So far we have assumed that every hidden layer is widened by the same factor. But in fact we can widen different hidden layers differently. This is useful, for example, in a Transformer where we may want to use a smaller d_{ffn} during tuning. If we are using Adam, as long

[20] In particular, we are not fixing the total training FLOPs when we scale, which requires understanding the tradeoff of different scale HPs. For example, when we transfer across batch size, we *fix* the number of steps of training (*not* the number of epochs), so that the total FLOPs scales linearly.

[21] There's also a literature on the proper initialization for training deep networks effectively (e.g. [5, 16, 25, 37, 56, 57, 63]), but they do not study the *transferability* per se. See Section 10.3

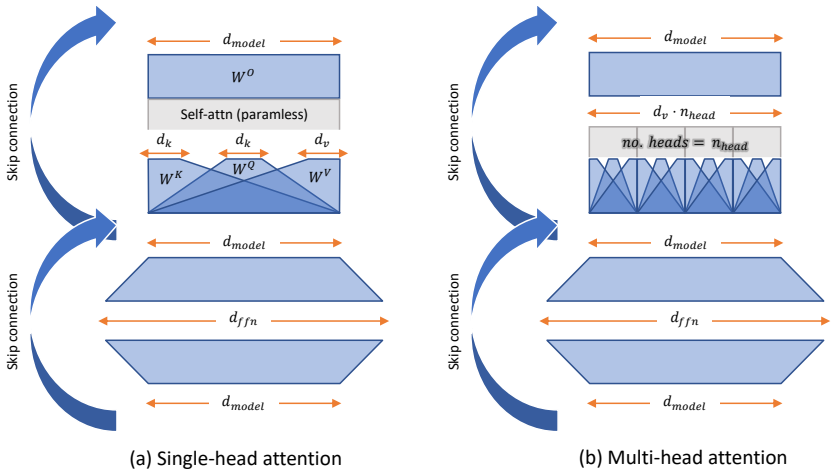


Figure 11: **Schematics of each Transformer layer.** Commonly, the key and value dimensions d_k and d_v are both set to d_{model}/n_{head}, and this is referred to as d_{head}.

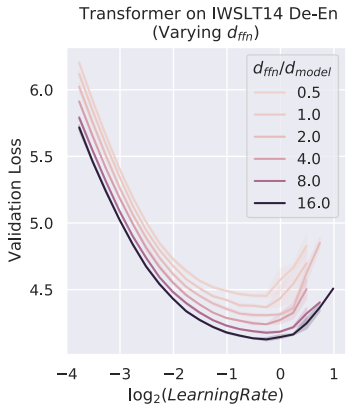


Figure 12: Learning rate landscape in μP is stable even if we vary d_{ffn} by a factor of 32, fixing d_{model}.

as the width of every layer still tends to infinity, we still obtain approximately the same limit[22], so the μTransfer remains theoretically justified.

See Fig. 12 for an empirical validation on IWSLT-14 using a Transformer.

Number of Attention Heads In attention-based models, one typically splits hidden size into multiple attention heads following $d_{model} = d_{head} \times n_{head}$. So far we have assumed d_{head} and d_{model} to be width, but it's possible and potentially advantageous to fix d_{head} and treat n_{head} as the width, or increasing both simultaneously. This allows our technique to handle many popular models, including GPT-3 [7], which scale up by fixing d_{head} and increasing n_{head}. See Fig. 13 for an empirical validation on Wikitext-2.

Varying Just the Width of Attention Heads A specific useful instance of varying width ratio is decoupling the key and value dimensions d_k and d_v and scaling d_k differently from (typically larger than) d_{model}/n_{head}. This works as long as we use $1/d$ scaled-attention as in Definition 4.1 (instead of $1/\sqrt{d}$ as is done commonly). When tuning on the small proxy model, if d_k is too small, the HP landscape can be quite noisy. Keeping d_k relatively large while shrinking all other dimensions solves this problem, while still obtaining significant speedup.

[22]This also applies for SGD, but we need more involved scaling to keep the limit approximately the same.

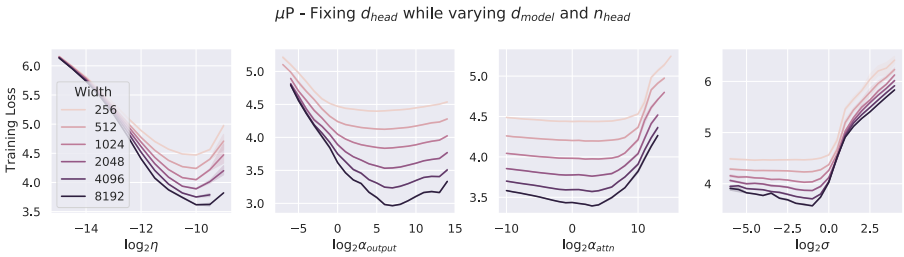


Figure 13: μTransfer across width when we fix d_{head} and vary d_{model} and n_{head}. $\alpha_{output}, \alpha_{attn}$ are multipliers for output and key weights, and σ is initialization standard deviation.

F Experimental Details

F.1 IWSLT

IWSLT14 De-En is a well-known machine translation benchmark. We use a Transformer implemented in `fairseq` [31] with a default $d_{model} = 1/4 d_{ffn} = 512$ and $d_k = d_q = d_v = {}^{d_{model}}/_{n_{head}} = 128$ (amounting to 40M parameters), which we denote as the *1x model*. For transfer, we tune on a proxy model with the same n_{head} but with d_{model} and other dimensions 4 times smaller; we will call this the *0.25x model* (but it has 4M parameters). All models are trained with Adam for 100 epochs and validated at the end of every epoch. We tune via random search the learning rate η, the output layer parameter multiplier α_{output}, and the attention key-projection weight multiplier α_{attn} following the grid

- η: $5 \times 10^{-4} \times 2^z$, where $z \in \{-1.5, -1.25, -1, ..., 1.25\}$
- α_{output}: 2^z, where $z \in \{-8, -7, -6, ..., 7\}$
- α_{attn}: 2^z, where $z \in \{-3, -2, -1, ..., 8\}$

F.2 WMT

We scale up to WMT14 En-De using the large Transformer from [47], with a $d_{model} = 1/4 d_{ffn} = 1024$ and $d_q = d_k = d_v = {}^{d_{model}}/_{n_{head}} = 64$. We use the exact same setup and reproduce their result as our baseline. Then, we build the proxy model by shrinking the target model's d_{model} from the original 1024 to 256, d_{ffn} from 4096 to 256 and n_{head} from 16 to 4. This reduces the total parameter count from 211M to 15M. We then perform the HP search on the proxy model and take the best according to validation loss, before testing on the target model. We tune via random search the learning rate η, the output layer parameter multiplier α_{output}, and the attention key-projection weight multiplier α_{attn} following the grid

- η: $6 \times 10^{-4} \times 2^z$, where $z \in \{-1.5, -1.25, -1, ..., 1.25\}$
- α_{output}: 2^z, where $z \in \{-8, -7, -6, ..., 7\}$
- α_{attn}: 2^z, where $z \in \{-3, -2, -1, ..., 8\}$

F.3 BERT

Details of BERT Prototype Our proxy model has 10 Transformer layers with $d_{model} = d_{ffn} = 256$. We also reduce the number of attention heads to 8 with a d_{head} of 32. We call it BERT Prototype since we can increase its width and depth according to our definitions to recover both BERT Base and BERT Large, which enables us to sweep HPs once and use for both models. Overall, BERT Prototype has 13M trainable parameters, a fraction of the 110M in BERT Base and the 350M in BERT Large.

Hyperparameters Tuned for Pretraining We tune the following HPs for pretraining: Adam learning rate η, embedding learning rate η_{emb}, output weight multiplier α_{output}, attention logits multiplier α_{attn}, layernorm gain multiplier $\alpha_{LN_{gain}}$, and bias multiplier α_{bias}.

We sample 256 combinations from the follow grid:

- η: $1 \times 10^{-4} \times 2^z$, where $z \in \{1.5, 2, 2.5, 3, 3.5\}$

- η_{emb}: $1 \times 10^{-4} \times 2^z$, where $z \in \{-1, -0.5, 0, 0.5, 1\}$
- α_{output}: 2^z, where $z \in \{2, 4, 6\}$
- α_{attn}: 2^z, where $z \in \{3, 3.5, 4, ..., 7\}$
- $\alpha_{LN_{gain}}$: 2^z, where $z \in \{8.5, 9, 9.5, 10, 10.5\}$
- α_{bias}: 2^z, where $z \in \{8.5, 9, 9.5, 10, 10.5\}$

The ranges are chosen to include the implicit choices of these HPs in SP BERT Large.

Finetuning Procedure and Hyperparameters We hand-pick the finetuning HPs after training the full-sized model. As regularization is an essential ingredient in successful finetuning, we do not transfer such HPs (at least via the suite of techniques presented in this work) (see Table 1). We focus on MNLI [49] and QQP, which are two representative tasks from GLUE [48]. Following [26], we used Adam [20] with a learning rate of 5×10^{-5} and a batch size of 64. The maximum number of epochs was set to 5. A linear learning rate decay schedule with warm-up of 0.1 was used. All the texts were tokenized using wordpieces and were chopped to spans no longer than 128 tokens.

F.4 GPT-3

Baseline 6.7B GPT-3 Transformer As the GPT-3 codebase has evolved since the publication of [7], we re-trained the 6.7B model from scratch to remove changes in our codebase as a possible confounder. The main differences to [7] are 1) a modified learning rate decay schedule, where the learning rate is decayed to zero at the end of training rather than being decayed to 0.1 of the initial value, and 2) use of relative attention in place of absolute attention. Unfortunately, after all experiments were finished, we found this re-run baseline used absolute attention instead of relative attention, while the μTransfer model still used relative attention.

Random Search using Reduced-Width Proxy Model In order to find a good set of hyperparameters for the μTransfer version of the 6.7B model, we performed a hyperparameter search over a reduced version of the model (i.e., the proxy model), where the width is set to 256 hidden units. This proxy model inherits changes from the evolved GPT-3 codebase: it uses relative [10] (instead of absolute) position encoding. Early on, we noted that on the proxy model, linear learning rate decay outperformed the default cosine schedule, so all subsequent experiments for the proxy models use a linear decay schedule. By Fig. 4, μTransferring this linear decay schedule to the full model should maintain such a performance advantage over the cosine schedule.

The hyperparameter search space consists of the following hyperparameters:

- **learning rate:** Sampled from $10^{\text{Uniform}(-4,-1)}$
- **initialization scale:** All the parameters are multiplied - sampled from $10^{\text{Uniform}(-1,1)}$
- **attention temperature:** Reciprocal of the multiplier applied to the input to attention softmax. Sampled from $4^{\text{Uniform}(-1,1)}$.
- **output temperature:** Reciprocal of the multiplier applied to the input to softmax that produces the distribution over output tokens. Sampled from $4^{\text{Uniform}(-1,1)}$.
- **embedding multiplier:** Scalar by which we multiply the output of the embedding layer. Sampled from $10^{\text{Uniform}(-1,1)}$.
- **relative position embedding multiplier:** Scalar by which we multiply vectors representing relative position. Sampled from $10^{\text{Uniform}(-1,1)}$.

In order to make the search more efficient we reduced the total number of training tokens. We hypothesized that tuning hyperparameters on a reduced total number of tokens does not significantly affect optimal hyperparameters. To verify, we trained two different horizons and compared the results. While the target model was to be trained on 300 billion tokens, we tuned the proxy model on only subsets consisting of 4 billion and 16 billion tokens. This impacts both the total training time and the length of the linear learning rate decay schedule. Other than hyperparameters explicitly listed above and the training horizon, the rest was the same as what we intended to use for the full width 6.7B training run.

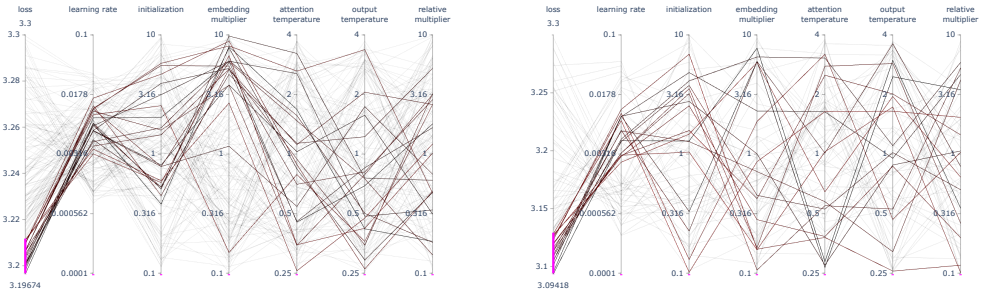


Figure 14: **Results of the random search over reduced-width GPT-3 proxy models** trained on 4 (left) and 16 (right) billion tokens. Only the best performing runs are highlighted.

Analyzing the Results of the Random Search We performed 467 training runs of the proxy model, out of which 350 were for 4 billion tokens (286 completed without diverging) and 117 for 16b tokens (80 completed without diverging). See Fig. 14 for summary of the results.

As suspected, we observed that the results are well-aligned for both 4 and 16 billion tokens versions. We observe learning rate and initialization scale impact the results the most. Based on the results we chose 0.006 for the former and 2.5 for the latter. Since most other hyperparameters appear to have negligible effect on performance, they were kept at their default values of 1, the only exception being the embedding scale, where higher values seem to perform better and it was therefore set to 10.

Training the μTransfer Model We encountered frequent divergences in our initial attempt to train the μTransfer model. We traced the issue back to underflow of FP16 tensors in the backwards pass and therefore switched to training the model in FP32. This allowed us to finish the training run without divergences. We hypothesize that the divergence issue is related to μTransfer picking more aggressive hyperparameters, for example a higher learning rate on linear weight tensors compared to the original model. In order to exclude code differences as a possible confounder, we re-trained GPT-3 6.7B from scratch using the original hyperparameters. The only difference compared to the version published in [7] is that the learning rate was decayed fully, whereas the learning rate of the model from [7] was only decayed to 10% of its starting value. The retrained model performs slightly worse than the original published in [7]. We suspect that this is because it made less progress during the last phase of training where the learning rate is close to zero. The training curves of the μTransfer model and the re-run of the original 6.7B can be seen in Fig. 15. Detailed evaluation results can be found in Table 10 and Table 11.

Ratio of Tuning Cost to Pretraining Cost in FLOPs can be approximated as

$$\frac{s(t_1 N_1 + t_2 N_2)}{ST} \approx 0.07$$

where

- $s = 40$ Million is number of parameters of the proxy model
- $S = 6.7$ Billion is number of parameters of the target model
- $t_1 = 4$ Billion is the number of training tokens for the short horizon HP search, and $N_1 = 350$ is the corresponding number of random HP search trials.
- $t_2 = 16$ Billion is the number of training tokens for the longer horizon HP search, and $N_1 = 117$ is the corresponding number of random HP search trials.
- $T = 300$ Billion is the number of training tokens for the 6.7B target model.

Here we are using the fact that the training FLOPs of a Transformer per token is roughly proportional to its number of parameters.

Table 10: **Full evaluation results of our GPT-3 6.7B models**: The new model tuned with μTransfer (marked *μP*), the original model from [7], and a re-training of this model from scratch with the original hyperparameter settings (marked *re-run*). The sampling-based evaluations shown here are a subset of the ones from [7]. Since the sampling-based evaluations are subject to high variance, Wikitext 103 and the LM1B benchmark have been added to help distinguish the relative performance of the μP and non-μP model. Note that Wikitext-103 [30] and the LM1B [9] benchmarks overlap with the training dataset. Accuracies and F1 scores have been multiplied by 100. The perplexities reported in this table are based on a custom BPE encoding and are not comparable to other results in the literature. The number k of examples in the context for each task is identical to [7].
Note: Zero-shot, One-Shot and Few-Shot refer to the number of additional query and answer pairs passed in the context when performing the sampling-based evaluations, not the "shots" involved in hyperparameter transfer.

			Zero-shot			One-shot			Few-shot		
Task	Split	Metric	μP	[7]	re-run	μP	[7]	re-run	μP	[7]	re-run
Validation dataset	valid	ce	**1.98**		2.03						
PTB	test	ppl	**11.4**		13.0						
Wikitext 103	test	ppl	**8.56**		9.13						
LM1B	test	ppl	**20.5**		21.7						
HellaSwag	dev	acc	**72.0**	67.4	66.7	**71.1**	66.5	65.9	**72.4**	67.3	66.4
LAMBADA	test	acc	**73.5**	70.3	70.8	**69.9**	65.4	64.8	74.7	**79.1**	77.1
StoryCloze	test	acc	**79.4**	77.7	77.3	**80.6**	78.7	78.3	**84.2**	81.2	81.1
NaturalQS	test	acc	**9.86**	5.79	7.20	**14.7**	9.78	10.6	**20.2**	17.0	15.7
TriviaQA	dev	acc	**47.0**	38.7	37.5	**50.4**	44.4	42.5	**55.5**	51.6	49.9
WebQS	test	acc	**11.3**	7.73	9.79	**20.2**	15.1	16.2	**33.0**	27.7	28.2
Ro→En 16	test	BLEU-sb	**26.9**	8.75	13.7	**36.5**	34.2	33.5	**38.2**	36.2	35.6
En→Ro 16	test	BLEU-sb	**18.1**	5.31	4.40	**21.0**	18.2	17.3	**22.0**	19.6	18.8
Fr→En 14	test	BLEU-sb	**29.8**	15.5	19.6	**31.7**	31.6	30.1	**38.0**	36.4	36.5
En→Fr 14	test	BLEU-sb	**29.6**	11.4	11.6	**28.8**	28.3	26.0	33.3	**33.3**	31.2
De→En 16	test	BLEU-sb	**31.7**	18.2	21.7	**33.3**	31.9	31.1	**38.9**	36.5	36.2
En→De 16	test	BLEU-sb	**23.1**	9.36	9.00	**24.6**	21.7	21.1	**27.6**	24.1	24.5
Winograd	test	acc	85.3	85.7	**86.8**	**84.6**	84.6	84.2	**86.4**	85.4	83.9
Winogrande	dev	acc	**66.8**	64.5	62.5	**67.6**	65.8	64.5	**71.0**	67.4	67.2
PIQA	dev	acc	**79.1**	78.0	78.0	**77.3**	76.3	76.9	**79.2**	77.8	77.7
ARC (Challenge)	test	acc	42.1	41.4	**42.5**	**44.0**	41.5	42.4	**43.8**	43.7	42.7
ARC (Easy)	test	acc	**64.3**	60.2	61.9	**65.3**	62.6	63.4	**67.3**	65.8	65.3
OpenBookQA	test	acc	**54.4**	50.4	52.6	**56.4**	53.0	52.8	**58.4**	55.2	54.4
Quac	dev	f1	**41.8**	36.1	38.2	**43.1**	39.0	39.5	**44.0**	39.9	39.9
RACE-h	test	acc	**45.0**	44.1	43.2	**44.9**	44.3	42.9	**45.2**	44.7	43.4
RACE-m	test	acc	**58.4**	54.4	54.0	**57.9**	54.7	53.8	**58.6**	55.4	55.4
SQuADv2	dev	f1	**59.9**	52.7	50.9	**64.9**	57.1	54.7	**68.9**	62.1	58.4
CoQA	dev	f1	**78.5**	72.8	72.9	**80.9**	75.1	74.4	**81.3**	77.3	75.4
DROP	dev	f1	17.1	17.0	**17.4**	23.3	**27.3**	25.7	**33.9**	29.7	28.7
BoolQ	dev	acc	**69.4**	65.4	60.9	**74.1**	68.7	65.0	**73.9**	70.0	69.7
CB	dev	acc	21.4	28.6	**37.5**	**60.7**	33.9	32.1	62.5	60.7	**66.1**
Copa	dev	acc	**82.0**	80.0	77.0	81.0	**82.0**	81.0	**88.0**	83.0	82.0
RTE	dev	acc	**55.2**	55.2	46.2	**61.0**	54.9	58.8	52.7	49.5	**59.9**
WiC	dev	acc	**0.**	**0.**	**0.**	50.0	50.3	**50.3**	50.5	**53.1**	51.3
ANLI R1	test	acc	**33.7**	32.3	33.4	**32.4**	31.6	31.7	30.9	**33.1**	30.7
ANLI R2	test	acc	**33.8**	33.5	33.0	**34.8**	33.9	33.7	**35.0**	33.3	32.2
ANLI R3	test	acc	32.7	**34.8**	33.4	**34.8**	33.1	33.3	**36.9**	33.9	32.3

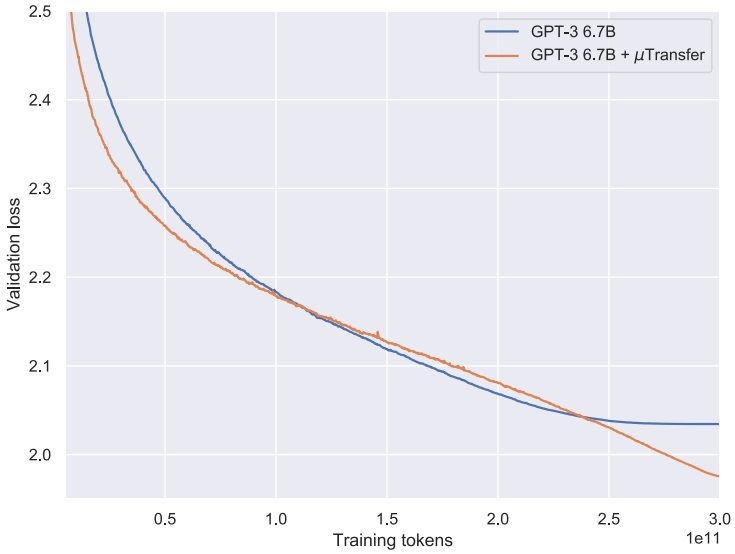


Figure 15: **The training curves of the GPT-3 6.7B model with μTransfer (orange) and a re-run with the original settings from [7] (blue).** The μTransfer model uses relative attention while the re-run uses absolute attention. In addition, the former was trained using FP32 activations and weights after initially encountering stability issues with the hyperparameters computed using μP, while the re-run used the original FP16 training. The μTransfer model seems to underperform in the middle of training, but achieves a much better final validation loss once the learning rate is fully decayed. While the original model uses a cosine schedule, the μTransfer model uses a linear learning rate decay schedule transferred from the proxy model.

G Additional Experiments

G.1 Experiments on ResNets

G.1.1 ResNet on CIFAR-10

Setup For this case we use Davidnet [2], a ResNet variant that trains quickly on CIFAR-10, so as to efficiently investigate its HP landscape. We train with SGD on CIFAR-10 for 10 epochs; all results are averaged over 15 random seeds. We use a width multiplier to identify models of different width, and a multiplier of 1 corresponds to the original model in [2]. We look at validation accuracy here as the model barely overfits, and our observations will hold for the training accuracy as well. We first conduct a learning rate sweep for models of different widths using SP; the result is shown in Fig. 16, on the left.

Hyperparameter Stability Note that the best model with a width multiplier of 8 under-performs that with a multiplier of 4. We run the same sweep with μP, along with a sweep of the output multiplier (α_{output}); the result is shown in Fig. 16, on the right. We notice that wider models always perform better under μP and that the optimal learning rate η and α_{output} are stable across width.

Hyperparameter Transfer Next, we perform a grid search for learning rate (η) and α_{output} on the 0.5x model for both SP and μP.[23] Then, we take the best combination and test on the 8x model, simulating how a practitioner might use μTransfer. The result is shown in Table 12, where μP outperforms SP by $0.43\% \pm .001\%$.

[23] Here we tune the 0.5x model instead of the 1x model to simulate the situation that one does "exploratory work" on the 1x model but, when scaling up, would like to tune faster by using a smaller proxy model.

Table 11: **Evaluation results comparing the GPT-3 6.7B model tuned with μTransfer against the twice-as-large GPT-3 13B model from [7].** The two models have similar performance on most of the evaluation tasks.

			Zero-shot		One-shot		Few-shot	
Task	Split	Metric	6.7B+μP	13B[7]	6.7B+μP	13B[7]	6.7B+μP	13B[7]
HellaSwag	dev	acc	**72.0**	70.9	**71.1**	70.0	**72.4**	71.3
LAMBADA	test	acc	**73.5**	72.5	**69.9**	69.0	74.7	**81.3**
StoryCloze	test	acc	79.4	**79.5**	**80.6**	79.7	**84.2**	83.0
NaturalQS	test	acc	**9.86**	7.84	**14.7**	13.7	20.2	**21.0**
TriviaQA	dev	acc	**47.0**	41.8	50.4	**51.3**	55.5	**57.5**
WebQS	test	acc	**11.3**	8.22	**20.2**	19.0	33.0	**33.5**
Ro→En 16	test	BLEU-sb	**26.9**	20.8	36.5	**36.7**	38.2	**38.4**
En→Ro 16	test	BLEU-sb	**18.1**	6.43	**21.0**	20.8	**22.0**	21.8
Fr→En 14	test	BLEU-sb	**29.8**	22.4	**31.7**	31.4	38.0	**38.3**
En→Fr 14	test	BLEU-sb	**29.6**	15.3	28.8	**30.1**	33.3	**35.5**
De→En 16	test	BLEU-sb	**31.7**	24.4	33.3	**34.5**	38.9	**39.1**
En→De 16	test	BLEU-sb	**23.1**	11.0	**24.6**	23.3	27.6	**27.7**
Winograd	test	acc	85.3	**87.9**	84.6	**86.1**	**86.4**	82.4
Winogrande	dev	acc	66.8	**67.9**	**67.6**	66.9	**71.0**	70.0
PIQA	dev	acc	**79.1**	78.5	77.3	**77.8**	79.2	**79.9**
ARC (Challenge)	test	acc	42.1	**43.7**	**44.0**	43.1	43.8	**44.8**
ARC (Easy)	test	acc	**64.3**	63.8	65.3	**66.8**	67.3	**69.1**
OpenBookQA	test	acc	54.4	**55.6**	**56.4**	55.8	58.4	**60.8**
Quac	dev	f1	**41.8**	38.4	**43.1**	40.6	**44.0**	40.9
RACE-h	test	acc	**45.0**	44.6	**44.9**	44.6	**45.2**	45.1
RACE-m	test	acc	**58.4**	56.7	**57.9**	56.9	**58.6**	58.1
SQuADv2	dev	f1	**59.9**	56.3	**64.9**	61.8	**68.9**	67.7
CoQA	dev	f1	**78.5**	76.3	**80.9**	77.9	**81.3**	79.9
DROP	dev	f1	17.1	**24.0**	23.3	**29.2**	**33.9**	32.3
BoolQ	dev	acc	**69.4**	66.2	**74.1**	69.0	**73.9**	70.2
CB	dev	acc	**21.4**	19.6	**60.7**	55.4	62.5	**66.1**
Copa	dev	acc	82.0	**84.0**	81.0	**86.0**	**88.0**	86.0
RTE	dev	acc	55.2	**62.8**	**61.0**	56.3	52.7	**60.6**
WiC	dev	acc	**0.**	**0.**	**50.0**	**50.0**	50.5	**51.1**
ANLI R1	test	acc	**33.7**	33.2	32.4	**32.7**	30.9	**33.3**
ANLI R2	test	acc	**33.8**	33.5	**34.8**	33.9	**35.0**	32.6
ANLI R3	test	acc	32.7	**34.4**	**34.8**	32.5	**36.9**	34.5

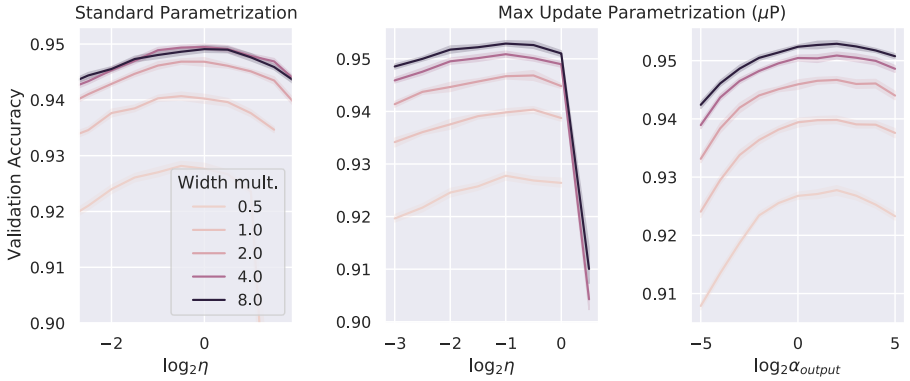


Figure 16: ResNet on CIFAR-10 for different widths (compared to a base network). On the **left**, the widest network SP underperforms; on the **right**, the μP network has a more consistent HP landscape and performs better. Both networks are tuned at the smallest width for the HP (η or α_{output}) not in the x-axis.

Table 12: ResNet on CIFAR10: Transferring the best learning rate (η) and α_{output} from widening factor 0.5 to 8; μP significantly outperforms SP given the same search grid. The best HPs are different as the models are parametrized to be identical at 1x width.[23]

Transfer Setup	Best η	Best α_{output}	Valid. Acc. (0.5x)	Valid. Acc. (8x)
SP	0.707	4	92.82%	94.86%
μP	0.5	4	92.78%	**95.29%**

G.1.2 Wide ResNet on ImageNet

Setup For this case we use Wide-Resnet, or WRN [62], a ResNet variant with more channels per layer, to further showcase μTransfer across width, i.e., number of channels. We train with SGD on ImageNet for 50 epochs following standard data augmentation procedures. We use a width multiplier to identify models of different width, and a multiplier of 1 corresponds to the original WRN-50-2-bottleneck in [62].

Hyperparameter Transfer We start with a proxy model with a width multiplier of 0.125 and tune several HPs using the following grid:

- η: $1 \times 2.048 \times 2^z$, where $z \in \{-5, -4, -3, ..., 4\}$
- α_{output}: 10×2^z, where $z \in \{-5, -4, -3, ..., 4\}$
- weight decay co-efficient γ: $3.05 \times 10^{-5} \times 2^z$, where $z \in \{-2, -1.5, -1, ..., 1.5\}$
- SGD momentum β: 0.875×2^z, where $z \in \{-2, -1.5, -1, ..., 1.5\}$

The grid is centered around the default HPs used by [1] for ResNet-50; while not expected to be competitive for WRN, they represent a reasonable starting point for our experiment.

We randomly sample 64 HP combinations from the grid and train for 50 epochs, before selecting the one with the highest top-1 validation accuracy. Then, we scale up the model following both μP and SP and run with the same HPs we just selected. The result is shown in Table 13, where μP outperforms SP by 0.41% in terms of top-1 validation accuracy.

Table 13: ResNet on ImageNet: Transferring the best learning rate (η), α_{output}, γ, and β from widening factor 0.125 to 1; μP significantly outperforms SP given the same search grid.

Transfer Setup	Best η	Best α_{output}	Best γ	Best β	Valid. Acc. (0.125x)	Valid. Acc. (1x)
SP	32.768	.625	.000015	.4375	58.12%	76.75%
μP	32.768	.625	.000015	.4375	58.12%	**77.16%**

G.2 Experiments on Transformers

G.2.1 Verifying Transfer across Batch Size, Sequence Length, and Training Time on Wikitext-2

See Fig. 19.

G.2.2 Post-Layernorm Transformers

Fig. 17 shows the transferability of learning rate, α_{output}, initialization standard deviation, and Adam β_2 across width, batch size, sequence length, and training steps for post-layernorm Transformers. However, in general, we find transfer across depth to be fragile.

G.2.3 Hyperparameter Instability of SP Transformers

Fig. 18 and Fig. 20 show the HP instability inherent in SP Transformers.

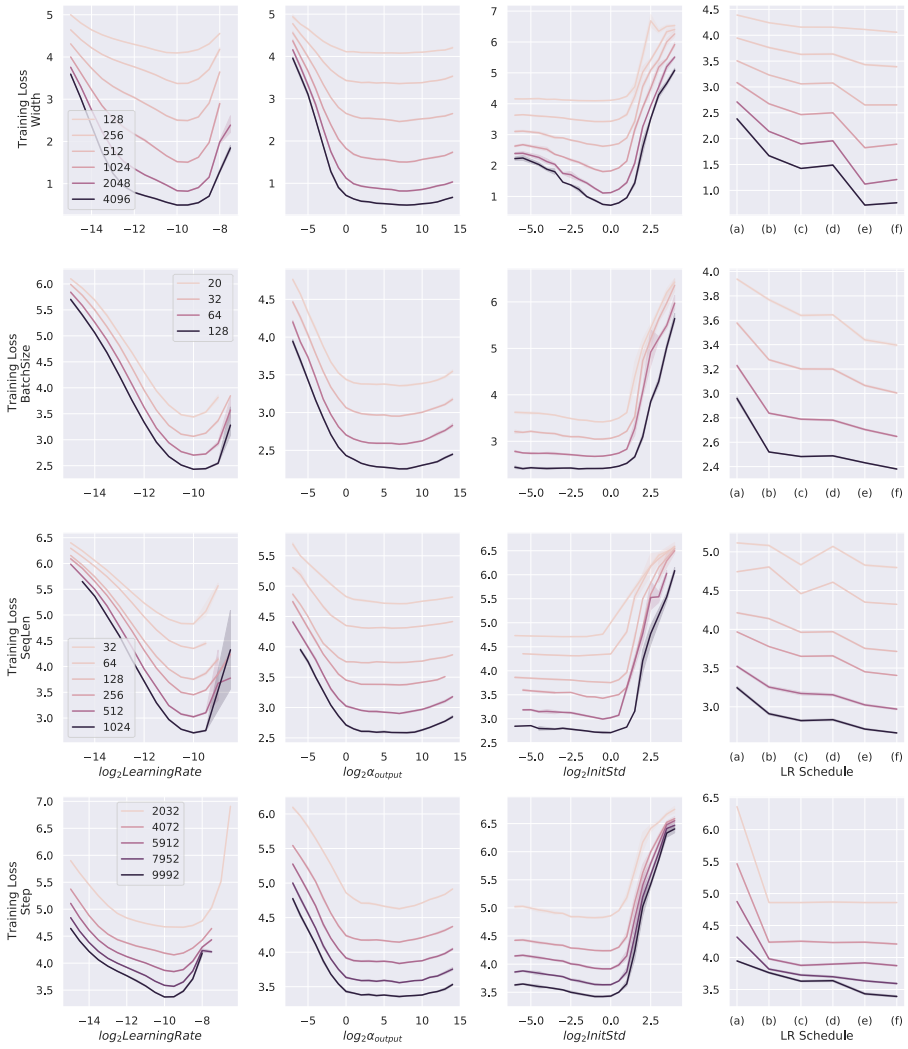


Figure 17: **Empirical validation of μTransfer for Post-LN Transformers.** Same setting as Fig. 4.

H Implementing μTransfer in a Jiffy

As we have shown, one can enable μTransfer by just reparametrizing the desired model in Maximal Update Parametrization (μP). While conceptually simple, switching from Standard Parametrization (SP) to μP can be error-prone, as popular deep learning frameworks are built around SP. We strive to build a tool that fulfills two goals:

1. Minimize code changes when switching to μP;
2. Keep model behavior invariant, under this switch, at a given *base model shape*.

By *model shape*, we mean the collection of dimensions of all parameters of the model. The latter goal, which we call *parametrization backward compatibility*, ensures that any code base works exactly as before at the base model shape, similar to Eq. (4), e.g. the loss at any time step remains exactly the same before and after the switch to μP. Of course, when widths start to differ from the base model shape, the model behavior necessarily changes so that HPs can be transferred.

There are two common approaches to setting the base model shape: 1) If one intends to tune a large target model, then the user can set the base model shape to be the shape of the target model (e.g. BERT-large or T5-large), so that the target model itself is in standard parametrization. Then one can tune a proxy model with e.g. $width = 124$ to obtain the optimal HPs for the target model. In addition, if one wishes to scale up further e.g. $width = 1024$, then these HPs remain optimal. 2)

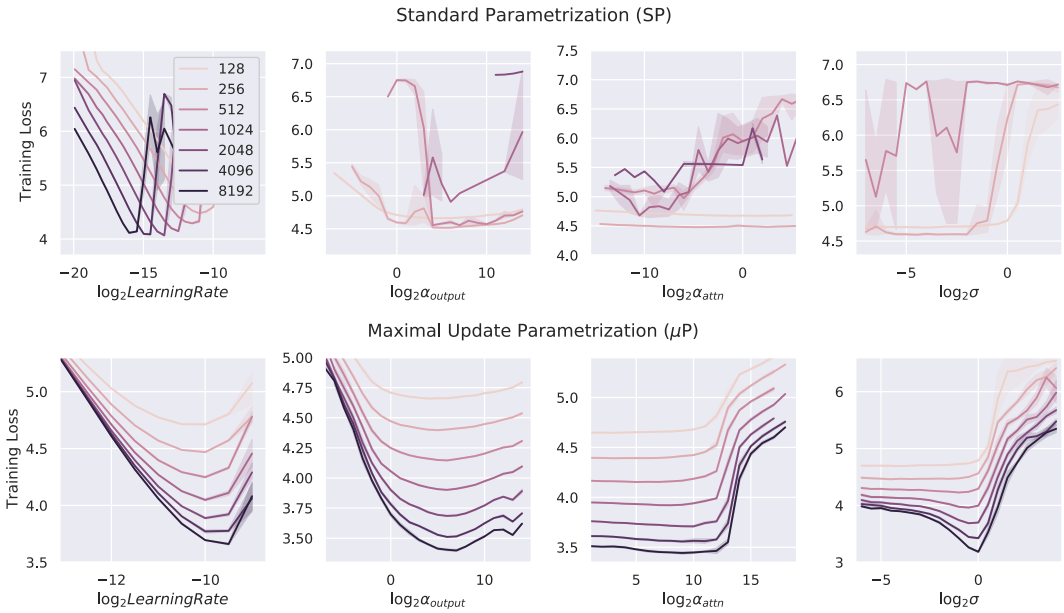


Figure 18: Post-layernorm Transformer with SP and μP on Wikitext-2. We sweep one HP across width (d_{model}) at a time while keeping the rest fixed; we also scale d_{head} linearly with d_{model} and fixing n_{head}. $\alpha_{output}, \alpha_{attn}$ are multipliers for output and key weights, and σ is initialization standard deviation. This yields unstable result for SP, as expected, where missing points/curves represent divergence; in μP, the optimal HP choices stabilize as width increases.

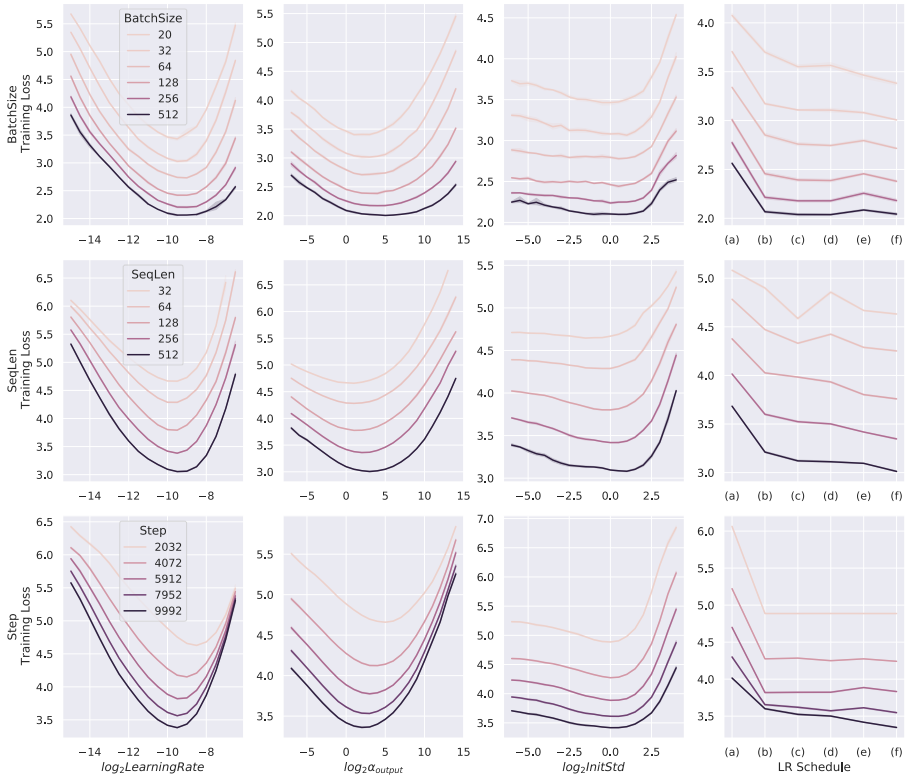


Figure 19: **Empirical validation of μTransfer across Batch Size, Sequence Length, and Training Time on pre-LN Transformers.** Same setting as Fig. 4. Despite some shift, the optimal HPs are roughly stable when transferring from batch size 32, sequence length 128, and 5000 training steps.

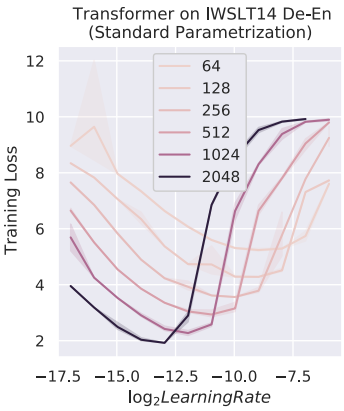


Figure 20: Learning rate landscape is highly unstable under standard parametrization in IWSLT.

If one has done exploration on a new idea with a small model and now wishes to scale up, reusing the HP found during this exploration, then one can set the base model shape to be the shape of the exploratory small model. Of course, in both scenarios, depth, batch size, and sequence lengths can be scaled up and down as well according to Fig. 19 (though note that currently we require users to recreate the base model shape at new depths, since the number of parameters now change with depth).

The `mup` Package We provide our tool as a Python package called `mup` designed to work with PyTorch. The following example illustrates the usage of our package.

```python
from mup.layers import MuReadout
from mup.shape import save_shapes, set_base_shapes
from mup.optim import MuSGD, MuAdam

class MyModel(nn.Module):
    def __init__(self, width, ...):
        ...
        ### In model definition, replace output layer with MuReadout
        # readout = nn.Linear(width, d_out)
        readout = MuReadout(width, d_out)
        ...
    def forward(self, ...):
        ...
        ### If using a transformer, make sure to use
        ### 1/d instead of 1/sqrt(d) attention scaling
        # attention_scores = query @ key.T / d**0.5
        attention_scores = query @ key.T / d
        ...

### Instantiate a base model
base_model = MyModel(width=1)

### Instantiate the target model (the model you actually want to train).
### This should be the same as the base model except the widths could be potentially different.
### In particular, base_model and model should have the same depth
model = MyModel(width=100)

### Set base shapes
set_base_shape(model, base_model)

### Alternatively, one can save the base model shapes in a file
# save_shapes(base_model, filename)
### and later set base shapes directly from the filename
# set_base_shape(model, filename)
### This is useful when one cannot fit both base_model and model in memory at the same time

for param in model.parameters():
    ### If initializing manually with fixed std or bounds,
    ### then replace with same function from mup.init
    # torch.nn.init.uniform_(param, -0.1, 0.1)
    mup.init.uniform_(param, -0.1, 0.1)
    ### Likewise, if using
    ### `xavier_uniform_, xavier_normal_, kaiming_uniform_, kaiming_normal_`
    ### from `torch.nn.init`, replace with the same functions from `mup.init`

### Use the optimizers from `mup.optim` instead of `torch.optim`
# optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
optimizer = MuSGD(model.parameters(), lr=0.1)

### Then just train normally
```

What Happens in the `mup` Package Under the hood, `mup` implements the μP formulation in Table 8. By invoking `set_base_shape(model, base_model)`, each parameter tensor `p` of `model` gets a `p.infshape` attribute that stores, for each of its dimensions, the corresponding base dimension and whether that dimension should be considered "infinite" (i.e. will be scaled up/down, e.g., d_{model} of a Transformer) or "finite" (i.e. will be fixed, e.g., vocabulary size). This information is used in the initializers and optimizers to automatically scale the parameters or learning rates to be compliant with μP. For example, by Table 8, the Adam learning rate of hidden weights `p` is calculated as η/`p.infshape.width_mult()`, where `p.infshape.width_mult()` essentially calculates $\frac{\texttt{fan\_in}}{\texttt{base\_fan\_in}}$.

I Reverse-μTransfer for Diagnosing Training Instability in Large Models

Large Transformers are famously fickle to train [24, 35]. We note that a possible source of this instability for larger transformers is the failure of naive hyperparameter transfer via the standard parametrization. This is certainly consistent with Fig. 1, which shows that the optimal learning rate for small Transformers can lead to trivial performance in large Transformers. We support this hypothesis further by *reverse-μTransferring the instability-inducing HPs from a large Transformer to a small one and replicating the training instability.* This is shown in Fig. 21.

Practically, this reverse-μTransfer technique can be used to diagnose or debug training instability problems of large models. We offer two case studies toward this claim.

1) When training transformers of width 8192 on Wikitext-2, we found certain HP combinations caused divergence in the middle of training. We reverse-μTransferred one such HP combination to a model of width 256 and replicated this divergence. By analyzing this small model's activations right before this divergence, we found that the cause is due to attention logits blowing up. Note this debugging session proceeded much more quickly than if we directly worked with the large model. Later we confirmed this is indeed the same cause of the width-8192 model's divergence.

2) A 6B-parameter language model (in standard parametrization) in a separate project experienced repeated blow-up in the middle of training. We reverse-μTransferred its hyperparameters to a smaller, 100M-parameter model and replicated the training instability. This was solved by a retuning of the small model via random search.

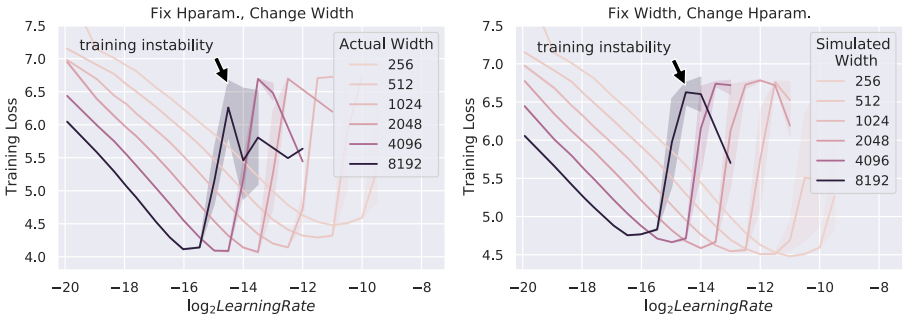


Figure 21: **Replicating training instability on a small Transformer by *reverse-μTransferring* hyperparameters.** These experiments concern 2-layer Transformers in Standard Parametrization (SP) on Wikitext-2, trained with Adam, where width is defined as $d_{model} = d_{ffn}$. (Left) LR-vs-loss for wider and wider Transformers. (Right) Likewise for *simulated width*: Here each point $(\log_2 \eta, loss)$ for simulated width n indicates the loss from training a width-256 μP Transformer with base width n and LR η (i.e. loosely speaking, it's using LR transferred from η in a width-n SP Transformer). **Takeaway:** The overall shapes of the curves are identical between the left and right plots[24]; in particular, a learning rate leads to instability in a wide model iff it does so when transferred back to a narrow model.

[24] Note that the curves on the left are "lower" than curves on the right. This just reflects the increasing capacity of wider models able to fit the training data better, so is orthogonal to our point.

J An Intuitive Introduction to the Theory of Maximal Update Parametrization

In what follows, we seek to describe useful intuitions and rules of thumb that would be helpful to practitioners and empirical researchers alike in figuring out what is the right neural network parametrization. The intuitions we shall describe regarding SGD can be made rigorous as in [53, 54]; those regarding Adam are new, and their formalization will be done in an upcoming paper.

First, we write down the most basic intuition regarding sums of many random elements, which will underlie all of the calculations that follow.

Law of Large Numbers (LLN) If $x_1, \dots, x_n, \dots$ "look like" random independent samples of a random variable X, then

$$\frac{1}{n}\sum_{i=1}^{n} x_i \to \mathbb{E}[X], \quad \text{as } n \to \infty.$$

Central Limit Theorem (CLT) In the same scenario as above,

$$\frac{1}{\sqrt{n}}\sum_{i=1}^{n}(x_i - \mathbb{E}[X]) \to \mathcal{N}(0, \sigma(X)), \quad \text{as } n \to \infty,$$

where $\sigma(X)$ is the standard deviation of the random variable X.

Of course, there are many subtleties one must resolve to make the statements above truly rigorous (e.g., what is the meaning of "look like"?), but as rules of thumb, they typically give the correct prediction.

In particular, here we want to note the following basic intuition regarding the size of a sum of x_i:

$$\text{when } n \text{ is large, } \sum_{i=1}^{n} x_i \text{ has typical size } \begin{cases} \Theta(n) & \text{if } \mathbb{E}[X] \neq 0 \\ \Theta(\sqrt{n}) & \text{otherwise} \end{cases}$$

Here, "typical size" can be taken to mean the size 99% of time. Again, we stress that this is a good rule of thumb that yields the correct prediction in the cases we are concerned with here; the rigorous versions of this will come from the Tensor Programs framework (e.g., [53]).

J.1 Behaviors of Gaussian Matrices vs Tensor Product Matrices

Central to the derivation of μP for any architecture are key insights on the behaviors of two kinds of random matrices: 1) iid Gaussian random matrix and 2) tensor product matrix (by which we mean a sum of outer products) and more generally what we call *nonlinear* tensor product matrix (see Eq. (7)). For example, a neural network, randomly initialized in the typical way, will have each weight matrix look like the former. However, every step of training by gradient descent adds a sum of outer products to this initial matrix, so that the *change in weights* constitute a tensor product matrix. For Adam, the change in weights is not a tensor product but a more general *nonlinear tensor product matrix (see Eq. (7))*. In this section, we will particularly focus on the *right scaling* for the entries of such matrices, leading to a discussion of *the right neural network parametrization* in the next section. We concentrate on the key heuristics but eschew burdensome rigor.

Key Insights Consider a random vector $v \in \mathbb{R}^n$ with approximately iid entries and a random matrix A of either size $n \times n$ or $1 \times n$, both having entries of size $\Theta(1)$.[25] In the context of deep learning, v for example can be an activation vector in an MLP, a Gaussian A the hidden weights at initialization, a (nonlinear) tensor product A the change in hidden weights due to training, and a vector A the readout layer weights. Then Av corresponds to a part of the next layer preactivation or the network output. To make sure the preactivations and the output don't blow up, we thus need to understand the scale of Av, especially in the general case where A is correlated with v.[26] This is summarized in Table 14, with the derivations below. Intuitively, a (nonlinear) tensor product or vector A will interact with a correlated v via Law of Large Numbers, hence the n-scaling, while a Gaussian A interacts with v via Central Limit Theorem, hence the $\sqrt{n}$-scaling.

[25] in the sense that the the variance of the entries are $\Theta(1)$

[26] Here "correlated" formally means v depends on $W^\top$ in a Tensor Program. This essentially captures all scenarios of "v correlated with W" that occurs in deep learning.

Table 14: Expected entry size of Av for different matrices A and vector v correlated with each other, both having entries of size $\Theta(1)$.

	Standard Gaussian $A \in \mathbb{R}^{n\times n}$	(Nonlinear) Tensor Product $A \in \mathbb{R}^{n\times n}$	Vector $A \in \mathbb{R}^{1\times n}$
Entry size of Av	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta(n)$

In the derivations below, we answer a slightly different but equivalent question of "how to scale A such that Av has entry size $\Theta(1)$?"

J.1.1 Preparation for the Derivations

By the results of [54], each (pre-)activation vector and its gradient vector in a multi-layer perceptron, at any time during training, have approximately iid coordinates in the large width limit,[27] and something similar can be said for more advanced networks such as ResNet and Transformers [28].

Definition J.1. We say any such vector $v \in \mathbb{R}^n$ *has* $\Theta(n^a)$*-sized coordinates*, or just $\Theta(n^a)$*-coordinates* for short, if $\|v\|^2/n = \Theta(n^{2a})$ as $n \to \infty$. Because, by the above discussion, the coordinates are roughly iid when n is large, this intuitively means that each entry of v has "typical size" $\Theta(n^a)$. We make similar definitions with Θ replaced by O and Ω.

Furthermore, to each such vector v with $\Theta(1)$-sized coordinates, we can associate a random variable Z^v, independent of n, that represents the coordinate distribution of v, in such a way that: If vector u is correlated with v, then Z^u will also be correlated with Z^v, and $\lim_{n\to\infty} v^\top u/n = \mathbb{E}\, Z^u Z^v$.

J.1.2 Linear Tensor Product Matrix (e.g. SGD Updates)

The case of (linear) tensor product matrix can be reduced to the outer product case by linearity. Given $u, v, x \in \mathbb{R}^n$ having approximately iid coordinates (of size $\Theta(1)$) like discussed above, we can form the outer product

$$A \overset{\text{def}}{=} u \otimes v/n = uv^\top/n, \tag{6}$$

which is the form of a single (batch size 1) gradient update to a weight matrix. Then, by Law of Large Numbers,

$$Ax = u\frac{v^\top x}{n} \approx cu, \quad \text{where} \quad c = \mathbb{E}\, Z^v Z^x.$$

So Ax also has approximately iid coordinates, distributed like $Z^{Ax} \overset{\text{def}}{=} Z^u\, \mathbb{E}\, Z^v Z^x$. Likewise, if A is a sum of outer products $A = \sum_{i=1}^k u^i \otimes v^i/n$, then

$$Ax = \sum_{i=1}^k u^i \frac{v^{i\top} x}{n}, \quad \text{with coordinates distributed as} \quad Z^{Ax} = \sum_{i=1}^k Z^{u^i}\, \mathbb{E}\, Z^{v^i} Z^x.$$

Notice that each coordinate of A has size $\Theta(1/n)$. The above reasoning shows that, in order for Ax to have coordinate size $\Theta(1)$ (assuming x does), then $\Theta(1/n)$ is the right coordinate size for A, in the general case that v^i and x are correlated (as is generically the case during gradient descent, with $A = \Delta W$ for some weights W and x being the previous activations).[29]

J.1.3 Nonlinear Tensor Product Matrix (e.g. Adam Updates)

When using Adam or another adaptive optimizer that normalizes the gradient coordinatewise before applying them, we need to modify our argument slightly to obtain the right coordinate size scaling of

[27] Our intuition here is derived from the assumption that width is much larger than training time; of course, as illustrated by our myriad experiments, these intuition are very useful even when this is not the case, such as when training to convergence.

[28] E.g. in a convnet, the (pre-)activations are iid across channels, but correlated across pixels

[29] In some corner cases when x is uncorrelated with v, then $v^\top x = \Theta(\sqrt{n})$ by Central Limit, so actually Ax has $\Theta(1/\sqrt{n})$-coordinates. However, this case does not come up much in the context of training neural networks.

the matrix. The gradient update A, after such normalization, will take the form of

$$A_{\alpha\beta} = \psi(u^1_\alpha, \dots, u^k_\alpha, v^1_\beta, \dots, v^k_\beta), \quad \text{for some } \psi : \mathbb{R}^{2k} \to \mathbb{R} \text{ and vectors } u^i, v^j \in \mathbb{R}^n. \tag{7}$$

We say a matrix of this form is a *nonlinear tensor product matrix*.

First, note the tensor product matrices (e.g. the form of SGD update) discussed previously (Eq. (6)) already takes this form, with $\psi(u^1_\alpha, \dots, u^k_\alpha, v^1_\beta, \dots, v^k_\beta) = n^{-1}(u^1_\alpha v^1_\beta + \cdots + u^k_\alpha v^k_\beta)$, so Eq. (7) is a strict generalization of linear tensor products. Next, for the example of Adam, each gradient update is μ/σ where μ (resp. σ^2) is the moving average of previous (unnormalized) gradients (resp. the coordinatewise square of the same).[30] If these unnormalized gradients are the outer products $u^1 \otimes v^1, \dots, u^k \otimes v^k$, then the update has coordinates

$$(\mu/\sigma)_{\alpha\beta} = \psi(u^1_\alpha, \dots, u^k_\alpha, v^1_\beta, \dots, v^k_\beta) \stackrel{\text{def}}{=} \sum_i \gamma_i u^i_\alpha v^i_\beta / \sqrt{\sum_i \omega_i (u^i_\alpha v^i_\beta)^2}, \tag{8}$$

where γ_i and ω_i are the weights involved in the moving averages.

Now suppose we have some $A \in \mathbb{R}^{n\times n}$ of the form Eq. (7), where $u^i, v^i \in \mathbb{R}^n$ have approximately iid coordinates (of size $\Theta(1)$), and $\psi = n^{-1}\bar{\psi}$ where $\bar{\psi}$ doesn't depend on n (in terms of Adam where $\bar{\psi}$ corresponds to the ψ of Eq. (8), this corresponds to using a learning rate of $1/n$). Then for $x \in \mathbb{R}^n$ having approximately iid coordinates of size $\Theta(1)$, by Law of Large Numbers,

$$(Ax)_\alpha = \frac{1}{n}\sum_{\beta=1}^n \bar{\psi}(u^1_\alpha, \dots, u^k_\alpha, v^1_\beta, \dots, v^k_\beta)x_\beta \approx \mathbb{E}\,\bar{\psi}(u^1_\alpha, \dots, u^k_\alpha, Z^{v^1}, \dots, Z^{v^k})Z^x \stackrel{\text{def}}{=} \Psi(u^1_\alpha, \dots, u^k_\alpha).$$

Here we made the obvious definition

$$\Psi : \mathbb{R}^k \to \mathbb{R}, \qquad \Psi(r_1, \dots, r_k) \stackrel{\text{def}}{=} \mathbb{E}\,\bar{\psi}(r_1, \dots, r_k, Z^{v^1}, \dots, Z^{v^k})Z^x.$$

Thus Ax also has approximately iid coordinates (of size $\Theta(1)$),

$$Z^{Ax} \stackrel{\text{def}}{=} \Psi(Z^{u^1}, \dots, Z^{u^k}).$$

For example, in the SGD example with $A = u \otimes v/n$ and $\bar{\psi}(u_\alpha, v_\beta) = u_\alpha v_\beta$, this formula gives $Z^{Ax} = \Psi(Z^u)$ where $\Psi(z) = z\,\mathbb{E}\,Z^v Z^x$, recovering the earlier derivation.

In any case, the point here is that A has coordinate size $\Theta(1/n)$, and this is the unique scaling that leads to Ax having coordinate size $\Theta(1)$.

J.1.4 Vector Case (e.g. Readout Layer)

The vector A case is similar to the tensor product cases above.

J.1.5 Gaussian Matrix (e.g. Hidden Weights Initialization)

Now consider the case where $A \in \mathbb{R}^{n\times n}$ is random Gaussian matrix with $A_{\alpha\beta} \sim \mathcal{N}(0, 1/n)$ and $x \in \mathbb{R}^n$ has approximately iid coordinates distributed like Z^x. In the context of neural network training, A should be thought of as a randomly initialized weight matrix, and x for example can be taken to be an activation vector in the first forward pass.

A Quick Intuition By standard random matrix theory, A has $\Theta(1)$ operator norm with high probability. Thus, with high probability, for any "typical" vector x, we expect $\|Ax\| = \Theta(\|x\|)$, even if x is correlated with A. If Ax's coordinates are "evenly distributed", then this would imply Ax has $\Theta(1)$-coordinates if x does. However, this is not so clear. Below we provide intuitions for why this would be the case.

[30] Adam also has bias correction for the moving averages which can be accomodated easily, but for simplicity we omit them here.

Intuition for Evenness of Coordinate Distribution If x is independent from A (or sufficiently uncorrelated), then each coordinate $(Ax)_\alpha$ has variance $\mathbb{E}(Z^x)^2 = \Theta(1)$ (so by definition has size $\Theta(1)$). Thus, here A having $\Theta(1/\sqrt{n})$-coordinates leads to Ax having $\Theta(1)$-coordinates, in contrast to the tensor product case above.

When x is correlated with A, it turns out the same scaling applies ($\Theta(1/\sqrt{n})$ is the unique scaling for A's entries such so that Ax has $\Theta(1)$ entries), but the reasoning is much more subtle: In the context of neural network training, it turns out all scenario where x is correlated with A can be reduced to the case where $x = \phi(A^\top y, \ldots)$ for some coordinatewise nonlinearity ϕ and some other vector $\mathbb{R}^n$.[31] Let's consider a very simple example with $x = A^\top \mathbf{1}$ for the all 1s vector $\mathbf{1} \in \mathbb{R}^n$ (which has coordinate size $\Theta(1)$ as can be checked easily). Then, for each index $\alpha \in [n]$, we can calculate

$$(AA^\top \mathbf{1})_\alpha = \sum_{\beta,\gamma} A_{\alpha\beta}A_{\gamma\beta} = \sum_\beta A_{\alpha\beta}^2 + \sum_\beta \sum_{\gamma\neq\alpha} A_{\alpha\beta}A_{\gamma\beta}.$$

Since $\mathbb{E}\, A_{\alpha\beta}^2 = 1/n$, by the Law of Large Number, the first sum $\sum_\beta A_{\alpha\beta}^2 \approx 1$. On the other hand, there are n summands of the form $\sum_{\gamma\neq\alpha} A_{\alpha\beta}A_{\gamma\beta}$, all iid with variance $\frac{n-1}{n^2} = \Theta(1/n)$. Thus by the Central Limit Theorem, we expect $\sum_\beta \sum_{\gamma\neq\alpha} A_{\alpha\beta}A_{\gamma\beta} \approx \mathcal{N}(0,1)$. Therefore, each coordinate of $(AA^\top \mathbf{1})_\alpha$ looks like $1 + \mathcal{N}(0,1) = \mathcal{N}(1,1)$ and thus has size $\Theta(1)$; again this is caused by A having $\Theta(1/\sqrt{n})$-coordinates.

This example can be generalized to more general x that is correlated with A, but the mathematics is quite involved. See [53] for more details.

J.2 Deriving μP for Any Architecture

Armed with the insight from the last section, we now outline the key steps to derive μP in Table 3 for any architecture. In practice, μP implies the following desiderata

Desiderata J.1. At any time during training

1. Every (pre)activation vector in a network should have $\Theta(1)$-sized coordinates[32]
2. Neural network output should be $O(1)$.
3. All parameters should be updated as much as possible (in terms of scaling in width) without leading to divergence.

Let's briefly justify these desiderata. For the desideratum 1, if the coordinates are $\omega(1)$ or $o(1)$, then for sufficiently wide networks their values will go out of floating point range. This problem is particularly acute for low-precision formats that are essential for training large models such as BERT or GPT. Moreover, a general nonlinearity is only well-behaved if its input is in a fixed range (although this is not a problem for homogeneous nonlinearities like relu). For example, for tanh nonlinearity, if the preactivation is vanishing $o(1)$, then $\tanh$ is essentially linear; if the preactivation is exploding $\omega(1)$, then the tanh gradient vanishes.

For the desideratum 2, a similar justification applies to the numerical fidelity of the loss function and loss derivative. Note that, with desideratum 3, this means the network output should be $\Theta(1)$ after training (but it can go to zero at initialization).

Finally, desideratum 3 means that 1) we are doing "maximal feature learning" [54] and 2) every parameter contribute meaningfully in the infinite-width limit. This ensures that learning rate "plays the same role" in the finite-width case as in the infinite-width limit. For example, it prevents the scenario where a weight matrix gets stuck at initialization in the limit for any learning rate (so learning rate does not matter) but evolves nontrivially in any finite-width network (so learning rate does matter).

These desiderata will essentially uniquely single out μP. More formally, μP is the unique parametrization that admits feature learning in all parameters of the neural network [54], and this property theoretically guarantees HP transfer across width (for sufficiently large width). However, for the sake

[31] This is because every "reasonable" deep learning computation can be expressed in a Tensor Program.

[32] In a convnet, a (pre-)activation vector corresponds to a single pixel across all channels; in general , we expect (pre-)activations are iid across channels, but correlated across pixels

of reaching a broader audience, we will focus more on the intuitive derivations from the desiderata rather than on this formal aspect.

Below, we first assume for simplicity that the width of every layer is n, and we focus only on dense weights. Later, we will discuss convolutions and varying the widths between layers.

J.2.1 μP Derivation From the Desiderata

Below, we will derive the μP formulation in Table 3. Tables 8 and 9 can be derived from Table 3 via the following equivalences, which can be easily derived via some simple calculations.

Lemma J.1. *Let $f_t(\xi)$ denote the neural network function after t steps of training (using any fixed sequence of batches), evaluated on input ξ. Consider a parameter tensor W with learning rate C, initialized as $W \sim \mathcal{N}(0, B^2)$, and with a multiplier A. Then for any $\theta > 0$, $f_t(\xi)$ stays fixed for all t and ξ if we set*

- *when the optimizer is SGD*

$$A \leftarrow A\theta,\ B \leftarrow B/\theta,\ C \leftarrow C/\theta^2$$

- *when the optimizer is Adam,*

$$A \leftarrow A\theta,\ B \leftarrow B/\theta,\ C \leftarrow C/\theta;$$

For example, for output weights, Table 3 has $A = 1$, $B = 1/\text{fan\_in}$, $C = \eta/\text{fan\_in}$ for SGD and Adam. Then taking $\theta = 1/\text{fan\_in}$, we get the entries in Table 8, with $A = 1/\text{fan\_in}$, $B = 1$, $C = \eta \cdot \text{fan\_in}$ for SGD and $C = \eta$ for Adam. Taking $\theta = 1/\sqrt{\text{fan\_in}}$ instead, we get the entries in Table 9, with $A = 1/\sqrt{\text{fan\_in}}$, $B = 1/\text{fan\_in}$, $C = \eta$ for SGD and $\eta/\sqrt{\text{fan\_in}}$ for Adam. Similar calculations hold for the input weights scaling in those tables, after taking into consideration that fan_in is considered a constant in terms of width for the input layer.

We proceed with the derivation of Table 3 below. Recall the definitions of $\Theta(n^a)$-*sized coordinates* or $\Theta(n^a)$-*coordinates* from Definition J.1.

Output Weights Suppose $W \in \mathbb{R}^{1\times n}$ is an output weight. By desideratum 1, the input x to W has $\Theta(1)$-sized coordinates. Thus W should have $\Theta(1/n)$-coordinates so that $|Wx| = O(1)$. We can initialize W with $\Theta(1/n)$-coordinates and scale its (per-layer) LR so that ΔW has $\Theta(1/n)$-coordinates as well. This means initializing $W_{\alpha\beta} \sim \mathcal{N}(0, \Theta(1/n^2))$ and use $\Theta(1/n)$ learning rate for both SGD and Adam.

Hidden Weights Consider a square weight matrix $W \in \mathbb{R}^{n\times n}$. Desiderata 1 guarantees that the input x to W has $\Theta(1)$-sized coordinates. Generally, x will be correlated with W. By Table 14, we can immediately derive

Initialization W should be randomly initialized with coordinate size $\Theta(1/\sqrt{n})$

LR The learning rate should be scaled so that ΔW has coordinate size $\Theta(1/n)$

so that $(W_0 + \Delta W)x$ is $\Theta(1)$ if x is, inductively satisfying desideratum 1. With Adam, this just means the per-layer LR is $\Theta(1/n)$. With SGD and the scaling of output layers above, we can calculate that the gradient of W has $\Theta(1/n)$-coordinates, so the $\Theta(1)$ SGD LR derived above suffices as well.

Input Weights Suppose $W \in \mathbb{R}^{n\times d}$ is an input weight. To satisfy desideratum 1 (i.e. for any input ξ, $W\xi$ should have $\Theta(1)$-coordinates), we want W to have $\Theta(1)$-coordinates. We can initialize W with $\Theta(1)$-coordinates and scale its (per-layer) LR so that ΔW has $\Theta(1)$-coordinates as well. This implies initialization variance of $\Theta(1)$ (or $\Theta(1/\text{fan\_in})$ since $\text{fan\_in} = \Theta(1)$ here) and Adam learning rate $\Theta(n)$. As above, we can calculate that the gradient of W has $\Theta(1/n)$-coordinates, so we want SGD learning rate $\Theta(n)$.

Biases Biases follow the same reasoning as input weights (just think of it as an input weight with input 1).

Attention Suppose the key dimension d_k is tending to infinity with width with number of heads n_{head} fixed. Then the key-query contraction $q^\top k \in \mathbb{R}$ scales like $\Theta(d_k)$ by Law of Large Numbers (instead of Central Limit Theorem because q and k are generally correlated) and desideratum 1, hence the $1/d_k$ we propose rather than $1/\sqrt{d_k}$.

Now suppose instead that n_{head} tends to infinity with width with d_k fixed. Let $K, Q \in \mathbb{R}^{N\times d_k \times n_{head}}, V \in \mathbb{R}^{N\times d_v \times n_{head}}$ be keys, queries, and values across all heads and tokens. Thinking of $N \times d_k$ as constants, we may view attention as a nonlinearity coordinatewise in the n_{head} dimension. Then it's clear that our parametrization described above already works.

Finally, we may freely let d_k and n_{head} both tend to infinity, and the above reasoning shows that our parametrization still works.

Changing Width Ratios As noted above, at any time in training, every (pre-)activation vector will have approximately iid coordinates (of order $\Theta(1)$ by desideratum 1). Another desideratum for μP is to ensure that this coordinate distribution (at any particular time) stays roughly invariant as widths increases. When all layer widths are tied, this is automatic if the other desiderata are satisfied, hence why we did not list this above.

When width ratios vary, this is not automatic. In this case, we need to choose whether to replace each n with fan-in or fan-out (or some function of them). Making the wrong choices will let the coordinate distributions vary with width ratios.

Obviously, we should replace n with fan-in for the output layers and with fan-out for the input layers since they are the only dimension scaling with n. For the hidden weights, we replace n with fan-in so that the forward pass is preserved. When using Adam (and assuming the initialization of W is quickly dominated by the change in W), this ensures that the (pre-)activation coordinate distributions are preserved at any time during training even if we vary widths in different layers differently. (For SGD this doesn't quite work in general because the varying width ratios change the gradient sizes of different layers differently, whereas Adam always normalizes the gradient coordinatewise).

Convolution A convolution weight tensor $W \in \mathbb{R}^{\text{fan\_out}\times\text{fan\_in}\times s_1 \times s_2}$ with kernel size $s_1 \times s_2$ can be thought of just as a $s_1 s_2 = \Theta(1)$-sized collection of $\text{fan\_out} \times \text{fan\_in}$ dense weights. Then all of our discussions above apply accordingly.

J.3 Why Other Parametrizations Cannot Admit Hyperparameter Transfer

Standard Parametrization (SP) SP doesn't work essentially because it leads to blow-up in the infinite-width limit.

1. For Adam with LR $\Theta(1)$, ΔW would have $\Theta(1)$-coordinates, causing preactivations to blow up like $\Theta(n)$ by Desideratum 1 and Table 14. We can avoid this blowup with LR $\Theta(1/n)$, but this induces a non-maximal feature learning limit, which, as we argue below, cannot transfer hyperparameters in all situations.
2. For SGD, the gradient of $\mathbb{R}^{n\times n}$ weight has $\Theta(1/\sqrt{n})$-coordinates, so $\Theta(1)$ learning rate would make preactivation scale like $\Theta(\sqrt{n})$ and hence blow up. If we use $\Theta(1/width)$ learning rate, then blow-up does not occur. However, this infinite-width limit is in the kernel regime [54] and thus does not allow HP transfer for the same reason that NTP below does not.

Neural Tangent Parametrization (NTP) We have concrete examples, e.g. Word2Vec in [54], where the NTK limit has trivial performance — so HPs have no effect at all — vastly outperformed by finite-width networks — where HPs matter. More importantly, wider does not always do better in NTP, especially in tasks where feature learning is crucial [54, 58]. So in the context of modern deep learning e.g. large language model pretraining, NTP (or SP with $\Theta(1/width)$ LR) does not make sense for wide neural networks.

Other Parametrizations Recall the *Dynamical Dichotomy Theorem* proven in [54], which says that any nontrivial stable "natural parametrization" (formally, "*abc-parametrization*," [54]) either admits a feature learning limit or a kernel limit, but not both.

Our argument above against SP and NTP will also work against any parametrization inducing a kernel limit. Therefore, it remains to ask, can other *feature learning* parametrizations transfer HPs?

We argue no. As shown in [54], any other feature learning parametrization differs from μP essentially only in that some parameters are not updated maximally. By [54, Sec 6.4], in the infinite-width limit, such parameters can be thought of as being fixed at initialization. Therefore, in such infinite-width limits, the learning rate of such parameters becomes useless. As such, we cannot hope for the HP landscape of the limit to reflect the HP landscape of finite-width neural networks.

μP is the unique feature learning parametrization that updates all parameters maximally, so that the learning rate of each parameter plays approximately the same role in finite-width neural networks as in the infinite-width limit. Consequently, the HP landscape of the μP limit should reflect the HP landscape of finite-width neural networks.

μTransfer: A technique for hyperparameter tuning of enormous neural networks

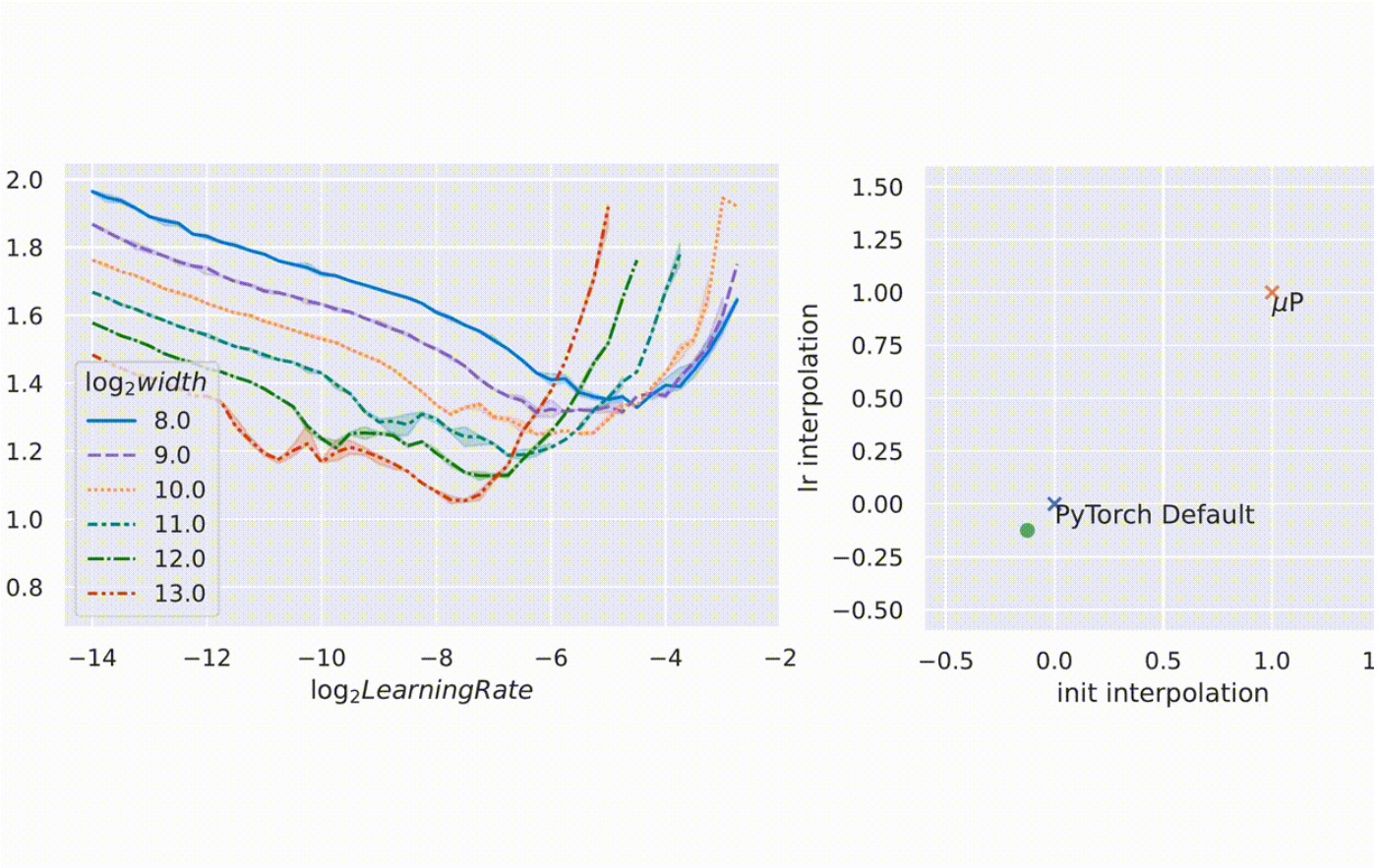
Published March 8, 2022

By Edward Hu, PhD Student; Greg Yang, Senior Researcher; Jianfeng Gao, Distinguished Scientist & Vice President

Research Area

Algorithms

Artificial intelligence



Great scientific achievements cannot be made by trial and error alone. Every launch in the space program is underpinned by centuries of fundamental research in aerodynamics, propulsion, and celestial bodies. In the same way, when it comes to building large-scale AI systems, fundamental research forms the theoretical insights that drastically reduce the amount of trial and error necessary and can prove very cost-effective.

In this post, we relay how our fundamental research enabled us, for the first time, to tune enormous neural networks that are too expensive to train more than once. We achieved this by showing that a particular parameterization preserves optimal hyperparameters across different model sizes. This is the µ-Parametrization (or *µP*, pronounced "myu-P") that we introduced in a previous paper, where we showed that it uniquely enables maximal feature learning in the infinite-width limit. In collaboration with researchers at OpenAI, we verified its practical advantage on a range of realistic scenarios, which we describe in our new paper, "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer."

By greatly reducing the need to guess which training hyperparameters to use, this technique can accelerate research on enormous neural networks, such as GPT-3

and potentially larger successors in the future. We also released a PyTorch package that facilitates the integration of our technique in existing models, available on the project GitHub page or by simply running `pip install mup`.

Read the paper › Download the code

"μP provides an impressive step toward removing some of the black magic from scaling up neural networks. It also provides a theoretically backed explanation of some tricks used by past work, like the T5 model. I believe both practitioners and researchers alike will find this work valuable."

— Colin Raffel, Assistant Professor of Computer Science, University of North Carolina at Chapel Hill and co-creator of T5

Scaling the initialization is easy, but scaling training is hard

Large neural networks are hard to train partly because we don't understand how their behavior changes as their size increases. Early work on deep learning, such as by Glorot & Bengio and He et al., generated useful heuristics that deep learning practitioners widely use today. In general, these heuristics try to keep the activation scales consistent at initialization. However, as training starts, this consistency breaks at different model widths, as illustrated on the left in Figure 1.

Unlike at random initialization, behavior during training is much harder to mathematically analyze. Our goal is to obtain a similar consistency so that as model width increases, the change in activation scales during training stay consistent and similar to initialization to avoid numerical overflow and underflow. Our solution, μP, achieves this goal, as seen on the right in Figure 1, which shows the stability of network activation scales for the first few steps of training across increasing model width.

Figure 1: In the default parameterization in PyTorch, the graph on the left, the activation scales diverge in width after one step of training. But in µP, the graph on the right, the activation scales change by a consistent amount regardless of width for any training step. The y-axis shows the change of network activation scales on a fixed input after t=0, 1, 2, 3, and 4 steps of training as the width of the model varies, which is shown along the x-axis.

Our parameterization, which maintains this consistency during training, follows two pieces of crucial insight. First, gradient updates behave differently from random weights when the width is large. This is because gradient updates are derived from data and contain correlations, whereas random initializations do not. Therefore, they need to be scaled differently. Second, parameters of different shapes also behave differently when the width is large. While we typically divide parameters into weights and biases, with the former being matrices and the latter vectors, some weights behave like vectors in the large-width setting. For example, the embedding matrix in a language model is of size *vocabsize x width*. While the width tends to infinity, *vocabsize* stays constant and finite. During matrix multiplication, the difference in behavior between summing along a finite dimension and an infinite one cannot be more different.

These insights, which we discuss in detail in a previous blog post, motivated us to develop µP. In fact, beyond just keeping the activation scale consistent throughout training, µP ensures that neural networks of different and sufficiently large widths behave similarly during training such that they *converge to* a desirable limit, which we call *the feature learning limit*.

A theory-guided approach to scaling width

Our theory of scaling enables a procedure to transfer training hyperparameters across model sizes. If, as discussed above, μP networks of different widths share similar training dynamics, they likely also share similar optimal hyperparameters. Consequently, we can simply apply the optimal hyperparameters of a small model directly onto a scaled-up version. We call this practical procedure *μTransfer*. If our hypothesis is correct, the training loss-hyperparameter curves for μP models of different widths would share a similar minimum.

Conversely, our reasoning suggests that no scaling rule of initialization and learning rate other than μP can achieve the same result. This is supported by the animation below. Here, we vary the parameterization by interpolating the initialization scaling and the learning rate scaling between PyTorch default and μP. As shown, μP is the only parameterization that preserves the optimal learning rate across width, achieves the best performance for the model with width 213 = 8192, and where wider models always do better for a given learning rate—that is, graphically, the curves don't intersect.

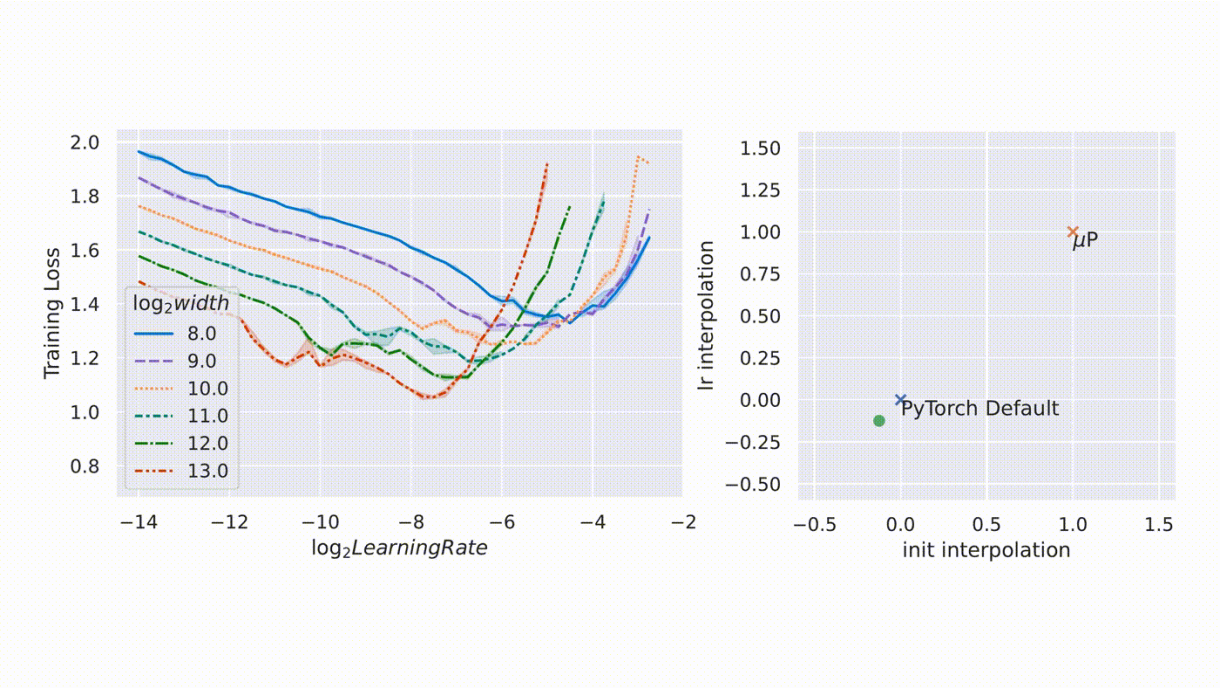


Figure 2: On the left, we train multilayer perceptrons (MLPs) of different widths (which correspond to the curves of different colors and patterns) with different learning rates (shown along the x-axis) on CIFAR10 and plot the training loss along the y-axis. On the right, the 2D plane of parameterizations is formed by interpolation of 1) the initialization scaling between PyTorch default and μP (x-axis), and 2) the learning rate scaling between PyTorch default and μP (y-axis). On this plane, PyTorch default is represented by (0, 0) and μP by (1, 1). The width-256 (log2(width) = 8) model is the same across all frames (except for random seed), but we widen models according to the parameterization represented by the dot on the right.

Building on the theoretical foundation of Tensor Programs, μTransfer works

automatically for advanced architectures, such as Transformer and ResNet. It can also simultaneously transfer a wide range of hyperparameters. Using Transformer as an example, we demonstrate in Figure 3 how the optima of key hyperparameters are stable across widths.

Figure 3: Transformers of different widths parameterized in μP and trained on WikiText-2. As we increase model width, the optimal learning rate, cross-entropy temperature, initialization scale, and learning rate schedule remain stable. We can meaningfully predict the optimal hyperparameters of a wider network by looking at those of a narrow one. In plot on the lower right, we tried the following learning rate schedules: (a) linear decay, (b) StepLR @ [5k, 8k] with a decay factor of 0.1, (c) StepLR @ [4k, 7k] with a decay factor of 0.3, (d) cosine annealing,(e) constant, and (f) inverse square-root decay.

> "I am excited about μP advancing our understanding of large models. μP's principled way of parameterizing the model and selecting the learning rate make it easier for anybody to scale the training of deep neural networks. Such an elegant combination of beautiful theory and practical impact."
>
> *— Johannes Gehrke, Technical Fellow, Lab Director of Research at Redmond, and CTO and Head of Machine Learning for the Intelligent Communications and Conversations Cloud (IC3)*

Beyond width: Empirical scaling of model

depth and more

Modern neural network scaling involves many more dimensions than just width. In our work, we also explore how μP can be applied to realistic training scenarios by combining it with simple heuristics for nonwidth dimensions. In Figure 4, we use the same transformer setup to show how the optimal learning rate remains stable within reasonable ranges of nonwidth dimensions. For hyperparameters other than learning rate, see Figure 19 in our paper.

Figure 4: Transformers of different sizes parameterized in μP and trained on Wikitext-2. Not only does the optimal learning rate transfer across width, as shown in Figure 3, it also empirically transfers across other scale dimensions—such as depth, batch size, and sequence length—across the ranges we tested here. This means we can combine our theoretically motivated transfer across width with the empirically verified one across other scale dimensions to obtain the practical procedure, μTransfer, to tune hyperparameters indirectly on a small model and transfer to a large one.

Testing μTransfer

Now that we have verified the transfer of individual hyperparameters, it is time to combine them in a more realistic scenario. In Figure 5, we compare μTransfer, which transfers tuned hyperparameters from a small proxy model, with directly tuning the large target model. In both cases, the tuning is done via random search. Figure 5 illustrates a Pareto frontier of the relative tuning compute budget compared with the tuned model quality (BLEU score) on IWSLT14 De-En, a

machine translation dataset. Across all compute budget levels, µTransfer is about an order of magnitude (in base 10) more compute-efficient for tuning. We expect this efficiency gap to dramatically grow as we move to larger target model sizes.

Figure 5: Across different tuning budgets, µTransfer dominates the baseline method of directly tuning the target model. As we train larger target models with billions of parameters, we expect the performance gap to widen, since the proxy model can remain small while still meaningfully predicting the optimal hyperparameters, as shown in Figures 3 and 4.

A glimpse of the future: µP + GPT-3

Before this work, the larger a model was, the less well-tuned we expected it to be due to the high cost of tuning. Therefore, we expected that the largest models could benefit the most from µTransfer, which is why we partnered with OpenAI to evaluate it on GPT-3.

After parameterizing a version of GPT-3 with relative attention in µP, we tuned a small proxy model with 40 million parameters before copying the best hyperparameter combination to the 6.7-billion parameter variant of GPT-3, as prescribed by µTransfer. The total compute used during this tuning stage was only 7 percent of the compute used in the pretraining of the final 6.7-billion model. This µTransferred model outperformed the model of the same size (with absolute attention) in the original GPT-3 paper. In fact, it performs similarly to the

model (with absolute attention) with double the parameter count from the same paper, as shown in Figure 6.

Figure 6: We applied μTransfer to GPT-3 6.7-billion parameter model with relative attention and obtained better results than the baseline with absolute attention used in the original GPT-3 paper, all while only spending 7 percent of the pretraining compute budget on tuning. The performance of this μTransfer 6.7-billion model is comparable to that of the 13-billion model (with absolute attention) in the original GPT-3 paper.

Implications for deep learning theory

As shown previously, μP gives a scaling rule which uniquely preserves the optimal hyperparameter combination across models of different widths in terms of training loss. Conversely, other scaling rules, like the default in PyTorch or the NTK parameterization studied in the theoretical literature, are looking at regions in the hyperparameter space farther and farther from the optimum as the network gets wider. In that regard, we believe that the feature learning limit of μP, rather than the NTK limit, is the most natural limit to study if our goal is to derive insights that are applicable to feature learning neural networks used in practice. As a result, more advanced theories on overparameterized neural networks should reproduce the feature learning limit of μP in the large width setting.

Theory of Tensor Programs

The advances described above are made possible by the theory of Tensor Programs (TPs) developed over the last several years. Just as autograd helps practitioners compute the gradient of any general computation graph, TP theory enables researchers to compute the limit of any general computation graph when

its matrix dimensions become large. Applied to the underlying graphs for neural network initialization, training, and inference, the TP technique yields fundamental theoretical results, such as the architectural universality of the Neural Network-Gaussian Process correspondence and the Dynamical Dichotomy theorem, in addition to deriving μP and the feature learning limit that led to μTransfer. Looking ahead, we believe extensions of TP theory to depth, batch size, and other scale dimensions hold the key to the reliable scaling of large models beyond width.

Applying μTransfer to your own models

Even though the math can be intuitive, we found that implementing μP (which enables μTransfer) from scratch can be error prone. This is similar to how autograd is tricky to implement from scratch even though the chain rule for taking derivatives is very straightforward. For this reason, we created the mup package to enable practitioners to easily implement μP in their own PyTorch models, just as how frameworks like PyTorch, TensorFlow, and JAX have enabled us to take autograd for granted. Please note that μTransfer works for models of any size, not just those with billions of parameters.

The journey has just begun

While our theory explains why models of different widths behave differently, more investigation is needed to build a theoretical understanding of the scaling of network depth and other scale dimensions. Many works have addressed the latter, such as the research on batch size by Shallue et al., Smith et al., and McCandlish et al., as well as research on neural language models in general by Rosenfield et al. and Kaplan et al. We believe μP can remove a confounding variable for such investigations. Furthermore, recent large-scale architectures often involve scale dimensions beyond those we have talked about in our work, such as the number of experts in a mixture-of-experts system. Another high-impact domain to which μP and μTransfer have not been applied is fine tuning a pretrained model. While feature learning is crucial in that domain, the need for regularization and the finite-width effect prove to be interesting challenges.

We firmly believe in fundamental research as a cost-effective complement to trial and error and plan to continue our work to derive more principled approaches to

large-scale machine learning. To learn about our other deep learning projects or opportunities to work with us and even help us expand μP, please go to our Deep Learning Group page.

Latest from

Edward Hu

PhD Student
Mila

View profile

Greg Yang

I am currently developing a framework called Tensor Programs for understanding large (wide) neural networks, and more generally, computational graphs such as commonly seen…

View profile

Jianfeng Gao

Distinguished Scientist & Vice President at Microsoft Research. IEEE Fellow. ACM Distinguished Member. I am leading the Deep Learning Group. The group's…

View profile

Related to this article

Research Groups

Deep Learning Group

Publications

Tensor Programs I: Wide Feedforward or Recurrent Neural Networks of Any Architecture are Gaussian Processes

Tensor Programs III: Neural Matrix Laws.

Tensor Programs II: Neural Tangent Kernel for Any Architecture.

Tensor Programs IV: Feature Learning in Infinite-Width Neural Networks

Tensor Programs IIb: Architectural Universality Of Neural Tangent Kernel Training Dynamics

Deep Residual Learning for Image Recognition

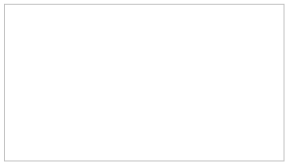
Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

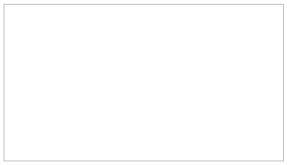
Up next

See all blog posts ›

On infinitely wide neural networks that exhibit feature learning ›



SOLOIST: Pairing transfer learning and machine teaching to advance task bots at scale ›



Adversarial robustness as a prior for better transfer learning ›

A deep generative model trifecta: Three advances that work towards harnessing large-scale power ›