Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer

Anonymous Author(s)
Affiliation
Address
email

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: 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 our approach on Transformer and ResNet. For example, by transferring pretraining HPs, we outperform BERT-large on MNLI and QQP, with a total tuning cost (in FLOPs) equivalent to pretraining BERT-large once.

1 Introduction

Hyperparameter tuning is critical to deep learning. Poorly chosen hyperparameters result in subpar performance and training instability. Many published baselines are hard to compare to one another due to varying degrees of hyperparameter 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.

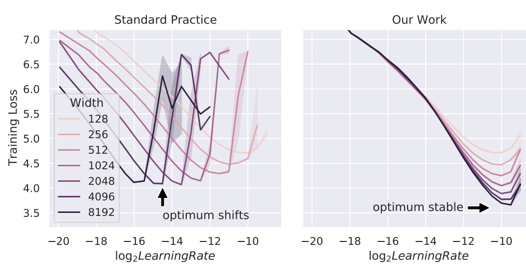


Figure 1: Training loss against learning rate on Transformers of varying d_{model} trained with Adam. Conventionally and in contrast with our technique, different widths do not share the same optimal hyperparameter; wider networks do not always perform better than narrower ones; in fact they underperform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

Recently, [40] 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*.[1] We leverage this new parametrization to *zero-shot transfer hyperparameters from small models to large models* in this work – that is, we obtain near optimal hyperparameters on a large model without directly tuning it at all! While practitioners have always guessed hyperparameters of large models from those of small models, the results are hit-or-miss at best, and this is 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 stable in standard parametrization (left). In addition to width, we empirically verify that, with a few caveats, hyperparameters can also be transferred across depth (in the main text) as well as batch size, language model sequence length, and training time (in the appendix). This reduces the tuning problem of an (arbitrarily) large model to that of a (fixed-sized)

[1] i.e., updates' effect on activations become roughly independent of width in the large width limit

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

Algorithm 1 Tuning a Large Target Model by 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 5.1. * 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*

small model. Our overall procedure, which we call *μTransfer*, is summarized in Algorithm 1 and Fig. 2, and the hyperparameters we cover are summarized in Tables 1 and 2.

There are several benefits to our approach: 1. **Speedup:** It provides massive speedup to the tuning of large models. For example, we are able to outperform published numbers of BERT-large [7], as measured on MNLI and QQP [37], purely by zero-shot hyperparameter transfer, with tuning cost approximately equal to only 1 BERT-large pretraining. For the largest models such as T5 [27] or GPT-3 [6], our approach renders hyperparameter tuning possible at all.[2] 2. **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 hyperparameters for all models in the family (but in general not for different data and/or tasks). For example, we will use this technique to tune BERT-base and BERT-large simultaneously. 3. **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 parallelization of tuning (and in the context of organizational compute clusters, better scheduling and utilization ratio).

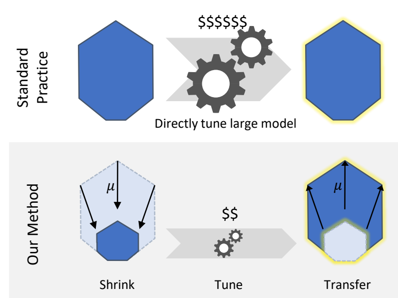


Figure 2: Illustration of μTransfer

Nevertheless, μTransfer still has several limitations. For example, while it is very effective for pretraining, it cannot transfer regularization hyperparameters, so it's generally not applicable to the finetuning of pretrained models. We discuss other limitations carefully in Section 5.1.

Our Contributions

- We demonstrate it is possible to zero-shot transfer near optimal hyperparameters to a large model from a small version;
- We propose a new hyperparameter tuning technique, *μTransfer*, for large neural networks based on this observation that provides massive speedup over conventional methods;
- We thoroughly verify our method on machine translation and large language models pretraining (in main text) as well as image classification (in appendix);
- We release a Pytorch[24] package for implementing μTransfer painlessly. A sketch of this package is given in Appendix E.

Terminologies Sometimes, to be less ambiguous, we will 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 hyperparameter tuning process. We will follow standard notation $d_{model}, d_{head} = d_k, d_v, n_{head}, d_{ffn}$ regarding dimensions in a Transformer; one can see Fig. 6 for a refresher.

[2] Note, again, that our tuning cost stays *fixed* even as the target model grows in size, so tuning T5-large would have the same cost as tuning BERT-large even though it is 4 times larger.

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

2 Parametrization Matters: A Primer

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

The Central Limit Theorem (CLT) says that, if $x_1, \ldots, x_n$ 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

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

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, $x_1, \ldots, x_n$ are akin to randomly initialized parameters of a width-n neural network, c is akin to a hyperparameter 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 hyperparameters correctly, then we can directly copy the optimal hyperparameters for a narrower network into a wide network and expect approximately optimal performance — this is the *hyperparameter transfer* we propose here. It turns out the Maximal Update Parametrization (μP) introduced in [40] 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) [40].[3] More concretely, as shown in [40] and Appendix G.3, in SP, the last layer is initialized too large and is updated disproportionally more as the model width increases, forcing a smaller learning rate to prevent a blow-up, consequently sacrificing performance.

3 Hyperparameters Don't Transfer Conventionally

In the community there seem to be conflicting assumptions about hyperparameter stability. *A priori*, models of different sizes don't have any reason to share the optimal hyperparameters. Indeed, papers aiming for state-of-the-art results often tune them separately. On the other hand, a nontrival fraction of papers in deep learning fixes all hyperparameters when comparing against baselines, which reflects an assumption that the optimal hyperparameters should be stable — not only between the same model of different sizes but also between models of different designs — so that such comparisons are fair. Here, we demonstrate hyperparameter *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.

[3]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 hyperparameter transfer because this limit is in kernel regime as shown in [40]. See Appendix G.3 for more discussions.

MLP with Standard Parametrization We start with a 2-hidden-layer MLP with activation function ϕ, using the standard parametrization[4] with LeCun initialization[5] 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.

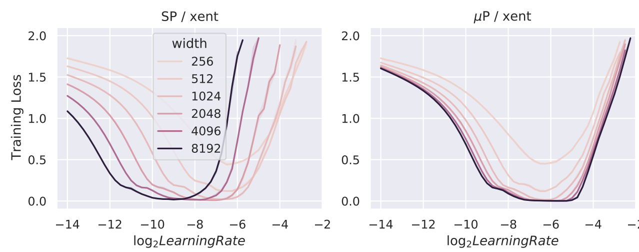


Figure 3: MLP width different hidden sizes trained for 20 epoch on CIFAR-10 using SGD. **Left** uses standard parametrization (SP); **right** uses maximal update parametrization (μP). μP networks exhibit better learning rate stability than their SP counterparts.

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 as d_{model}, with $d_k = d_q = d_v = {}^{d_{model}}\!/_{n_{heads}}$ and $d_{ffn} = 4d_{model}$. The models are trained on wikitext-2 for 5 epochs. In Fig. 12 in the appendix we also show the instability of initialization scale and other hyperparameters.

4 Unlocking Zero-Shot Hyperparameter Transfer with μP

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

MLP with μP The *basic form* of μP for the MLP in Section 3 is

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

Here we highlighted in purple the differences in the two parametrizations, namely the first layer initialization and the explicit multipliers in front of first layer weights, biases, and the output in μP. 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, a particularly useful instance of this is 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 $\tilde{n} \overset{\text{def}}{=} n/n_0$,

$$f(\xi) = {}^1\!/_{\sqrt{\tilde{n}}} W^{3\top}\phi(W^{2\top}\phi(\sqrt{\tilde{n}}W^{1\top}\xi + \sqrt{\tilde{n}}b^1) + \sqrt{\tilde{n}}b^2)$$
$$\text{with init.}\quad W^1 \sim \mathcal{N}(0, {}^1\!/_{\tilde{n}} \cdot {}^1\!/_{d_{in}}),\ W^{\{2,3\}} \sim \mathcal{N}(0, {}^1\!/_n),\ b^{\{1,2\}} = 0, \tag{4}$$

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 away 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 than SP as one varies n. As we will see, this deviation is crucial for hyperparameter 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

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

[5] 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[40] and SP for General Neural Networks, Basic Form.** This basic form emphasizes 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). Notations: 1) η is the "master" learning rate. 2) The fan_out of a bias vector is its dimension (whereas fan_in is 1). 3) *Multiplier* means explicit multipliers of the parameter, such as in Eq. (3). 4) 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, Adadelta, etc). Transformer μP requires one more modification ($1/d$ attention instead of $1/\sqrt{d}$); see Definition 4.1.

	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	η	η	η
Adam LR	$\eta/\sqrt{\text{fan\_out}}$ (η)	$\eta/\sqrt{\text{fan\_in}}$ (η)	$\eta/\text{fan\_in}$ (η)

means that, the best learning rate for a width-128 network is also best for a width-8192 network in μP — i.e. hyperparameter transfer *works* — but not for SP. In addition, we observe performance for a fixed learning rate always increases 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.

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.[6]

The results are shown on the right in Fig. 1, where the optimal learning rate is stable, and the performance improves monotonically as width increases.

5 Which Hyperparameters Can Be μTransferred?

In this section, we explore how common hyperparameters 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 hyperparameters.

Those in the first category transfer across width, as theoretically justified above in Section 2, while we empirically explore the transfer across the other dimensions in the third category, in order to push the practicality and generality of our technique. 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 B.2. This will be very useful in practice. For the second category, the amount of regularization 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 hyperparameters in more detail in Appendix B.1.

5.1 Empirical Validation and Limitations

Our empirical investigations focus on Transformers (here) and ResNet (in Appendix D.1.1), the most popular backbones of deep learning models today. We train a 2-layer pre-layernorm μP[7] Transformer

[6] 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 G.2.1 for a more in-depth discussion.

[7] "2 layers" means the model has 2 self-attention blocks. To compare with SP Transformer, see Fig. 12.

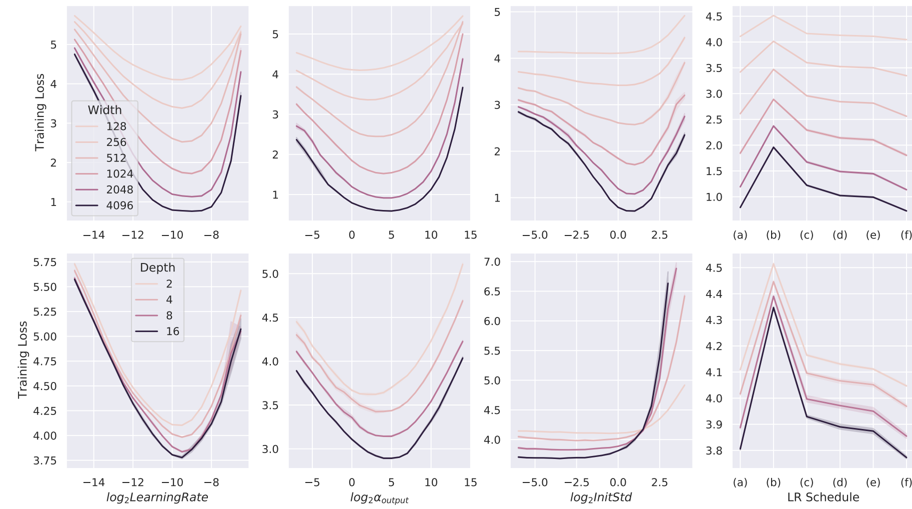


Figure 4: **Empirical validation of the stability of four representative hyperparameters on pre-LN Transformers in μP**: learning rate, last layer weight multiplier α_{output}, weight initialization standard deviation, and learning rate schedule. We use the following learning rate schedules: (a) constant; (b) linear decay; (c) StepLR @ [5k, 8k] with a decay factor of 0.1; (d) StepLR @ [4k, 7k] with a decay factor of 0.3; (e) cosine annealing; (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 32, LR schedule constant. We sweep a particular hyperparameter, corresponding to each column, while fixing all others constant. See Section 5.1 for discussion of these results.

with 4 attention heads on Wikitext-2. We sweep one of four hyperparameters (learning rate, output weight multiplier α_{output}, initialization standard deviation, and learning rate schedule) while fixing the others and sweeping along width and depth (with additional results in Fig. 10 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, hyperparameters 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, with some caveats discussed below. 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. 11),[8] the transfer across depth only works for pre-layernorm Transformer. Nevertheless, in practice (e.g. our results in Section 6.2.1) we find that fixing initialization standard deviation while tuning other hyperparameters works well when transferring across depth.

6 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 hyperparameter tuning only on a smaller proxy model, test the obtained hyperparameters on the large target model directly, and compare against baselines tuned using the target model. We seek to answer the question: Can μTransfer make hyperparameter 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 D.1. All of our experiments are run on V100 GPUs.

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

Table 4: **Transformer on IWSLT14 De-En.** 1x and 0.25x refers to scaling of width only. Compared to traditional tuning ("Tuning on 1x"), hyperparameter transfer 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 hyperparameter random search.

Setup	Total Compute	#Samples	Val. BLEU Percentiles 25	50	75	100
`fairseq`[22] 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**

6.1 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` [22] with 40M parameters, which we denote as the *1x model*.[9] 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 C.1.

We compare transferring from the 0.25x model with tuning the 1x model while controlling the total tuning budget in FLOPs.[10] To improve the reproducibility of our result: 1) we repeat the entire hyperparameter 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; 2) we evaluate each selected hyperparameter combination using 5 random initializations and report the mean performance.[11]

We pick the hyperparameter combination that achieves the lowest validation loss[12] 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.

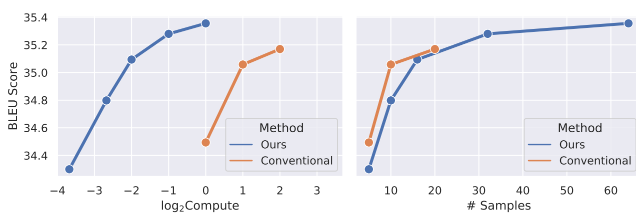


Figure 5: **Efficiency-Performance Pareto frontier** of μTransfer compared to conventional tuning, on IWSLT Transformer, using random hyperparameter search as the base method. We plot the *median* BLEU score over 25 trials (Left) against relative compute budget in log scale and (Right) against number of hyperparameter samples taken. While with the same number of samples, μTransfer slighly 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.

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. 5(left), where we repeat the above experiment with different compute budgets. Evidently, our approach completely dominates conventional tuning.

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

[10] Ideally we would like to measure the wall clock time of tuning. But for smaller models such as the IWSLT Transformer proxy models here, CUDA is poorly optimized, so wall clock time scaling would not reflect the scaling for larger models like BERT. Thus, we measure in FLOPs, which is less dependent on model size.

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

[12] We find this provides more reliable result than selecting for the best BLEU score because loss is "smoother."

Sample Quality of Proxy Model vs Target Model The Pareto frontier in Fig. 5(right) suggests that, given a fixed number of random *samples* from the hyperparameter 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 [40].[13] With few samples, this noise can distort the random hyperparameter search, but with more samples, this noise is suppressed.[14]

6.2 Transformer on WMT14 En-De

We scale up to WMT14 En-De using the large (post-layernorm) Transformer from [35] with 211M parameters. We tune on a proxy model with 15M parameters by shrinking d_{model}, d_{ffn}, and n_{heads}. 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 C.2. The result is shown in Table 5: While random search with 3 hyperparameter samples far underperforms the `fairseq` default, we are able to match it via transfer using the same tuning budget.

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 hyperparameter searches.

Setup	Total Compute	#Samples	Val. BLEU Percentiles Worst	Median	Best
`fairseq`[22] 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**

6.2.1 BERT

Finally, we consider large-scale language model pretraining where hyperparameter tuning is known to be challenging. Using Megatron (pre-layernorm) BERT [30] as a baseline, we hope to recover the performance of the published hyperparameters 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 5 and validated in Fig. 4. We use a batch size of 256 for all runs and follow the standard finetuning procedures. For more details on BERT-prototype, what hyperparameters we tune, and how we finetune the trained models, see Appendix C.3.

During hyperparameter 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 C.3. The results are shown in Table 6. Notice that on BERT-large, we obtain sizeable improvement over the pretuned Megatron BERT-large baseline.

7 Related Works

Hyperparameter Tuning Many have sought to speedup hyperparameter tuning beyond the simple grid or random search, such as via Bayesian optimization [32, 33] or multi-arm bandits [13, 16]. 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 work is complementary to the above, as they can be used to tune the proxy model. it is only for scientific reasons that we primarily did random search throughout this work.

Hyperparameter Transfer Many previous works explored transfer learning of hyperparameter tuning (e.g. [10, 25, 34, 43]). However, to the best of our knowledge, our work is the first to explore

[13]More precisely, the proxy and the target models are both "estimators" of their common *infinite-width limit*, but the former is more noisy than the latter, with width akin to the number of items in an average.

[14]but perhaps not completely, as the narrow proxy models may be biased estimators, i.e. the optimal hyperparameters might differ slightly from the wide model as seen in Fig. 4.

Table 6: **BERT pretraining.** Hyperparameter 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**

zero-shot hyperparameter transfer. In addition, we focus on transferring across model scale rather than between different tasks or datasets. Some algorithms like Hyperband [17] can leverage cheap estimates of hyperparameter 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.

Previously Proposed Scaling Rules of Hyperparameters [9, 21, 29, 31] investigated the right way to scale learning rate with batch size, with sometimes conflicting proposals. which we summarize in Appendix A. [23] studied how learning rate (and batch size) should scale with width for MLPs and CNNs trained with SGD in NTK or standard parametrizations. We provide a detailed comparison of our work with theirs in Appendix A.

Many previous works proposed different initialization or parametrizations with favorable properties, such as better stability for training deep neural networks [5, 8, 11, 19, 28, 41, 42, 45]. Our work differs from these in that we focus on the transferability of optimal hyperparameters from small models to large models in the same parametrization.

8 Conclusion

Leveraging the discovery of a feature learning neural network infinite-width limit, we hypothesized and verified that the hyperparameter 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 transfering the hyperparameters to the full model.

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 hyperparameter 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 consider if there's a way to transfer regularization hyperparameters as a function of both the model size and data size.

Broader Impact Our work makes hyperparameter tuning of large models more efficient. This enables large models to be better tuned given the same compute budget, thereby increasing the performance per cost. Organizations large and small can focus their research on small models and scale up only once with reasonable confidence that the training would go well. We do not foresee any direct negative societal impact.

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. arXiv: 2003.04887.

[6] 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.

[7] 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. arXiv: 1810.04805 version: 2.

[8] 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.

[9] 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. arXiv: 1705.08741.

[10] 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.

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

[12] 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. 00000 arXiv: 1806.07572.

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

[14] 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. arXiv: 2001.08361.

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

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

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

[18] 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`.

[19] 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`. arXiv: 2004.08249.

[20] 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`.

[21] 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`. arXiv: 1812.06162.

[22] 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.

[23] 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`.

[24] 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`.

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

[26] 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`.

[27] 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`. arXiv: 1910.10683.

[28] 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`. arXiv: 1611.01232.

[29] 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`. arXiv: 1811.03600.

[30] 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`.

[31] 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`. arXiv: 1711.00489.

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

[33] 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.

[34] 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`.

[35] 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`.

[36] 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.

[37] 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, 2019.

[38] 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`.

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

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

[41] 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-`.

[42] 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`. 00000 arXiv: 1712.08969.

[43] 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`. ISSN: 1938-7228.

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

[45] 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`.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section ?.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] . See abstract and Section 1.
   (b) Did you describe the limitations of your work? [Yes] . See Section 5.1 and Section 8.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] . See Section 8.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] .
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] Our main experiments are quite expensive and depend on details of our internal cluster, so we do not provide code to exactly reproduce it. However we include a package so that anyone can use μTransfer and describe all the details to reproduce it.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] . See Section 6 and Appendix C.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] . See Section 6.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] . We used Megatron, fairseq, pytorch and cited all of them.
   (b) Did you mention the license of the assets? [Yes] . In the references.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] . Yes, we are releasing a new Pytorch package.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Detailed Discussions on Related Works

A.1 Hyperparameter Tuning

Many have sought to speedup hyperparameter tuning beyond the simple grid or random search. Snoek et al. [32] treated hyperparameter tuning as an optimization process and used Bayesian optimization by treating the performance of each hyperparameter combination as a sample from a Gaussian process (GP). Snoek et al. [33] 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 [13, 16]. 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 hyperparameter 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 hyperparameter 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.

A.2 Previously Proposed Scaling Rules of Hyperparameters

(Learning Rate, Batch Size) Scaling [31] proposed to scale learning rate with batch size while fixing the total epochs of training; [9] proposed to scale learning rate as $\sqrt{batchsize}$ while fixing the total number of steps of training. However, [29] showed that there's no consistent (learning rate, batch size) scaling law across a range of dataset and models. Later, [21] 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.[15] This supports our results here as well as the empirical results in [29, fig 8].

Learning Rate Scaling with Width Assuming that the optimal learning rate should scale with batch size following [31], [23] empirically investigated how the "noise ratio" $LR/batchsize$ scales with width for MLP and CNNs in NTK parametrization (NTP) or standard parametrization (NTP) trained with SGD. 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.

Here in our work, Fig. 3 contradicts their results on SP MLP by showing the optimal learning rate (fixing batch size) shifts with width. We believe this difference is 1) due to their erroneous assumption that optimal learning rate scales with batch size (as debunked by [21, 29]) and 2) because their SP experiments were done by fixing the learning rate and only sweeping batch size.

Furthermore, Fig. 1 clearly shows the optimal learning rate is *not* constant in SP for transformers (trained with Adam). Other differences in our works include our applicability to 1) networks with normalization, 2) Adam and other adaptive optimizers, 3) our empirical validation of transfer across depth and sequence length, and 4) explicit validation of tuning via hyperparameter transfer on large models like BERT-large.

Finally, as argued in [40] and Appendix G.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 [40].

Input Layer Parametrization While typically, the input layer is initialized with fanin initialization, in language models where the input and output layers are shared (corresponding to word embeddings), it can actually be more natural to use a fanout initialization (corresponding to fanin initialization of

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

the output layer). In fact, we found that `fairseq` [22] by default actually implements our proposed input layer parametrization (both the fanout initialization and the $\sqrt{\text{fan\_out}}$ multiplier).[16]

From the Theory of Infinite-Width to the Practice of Finite-Width Neural Networks and Back [40] 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 [12] (which gives rise to the NTK limit) that does not learn features in the limit. Based on this theoretical insight, in Appendix G.3, we argue that μP should also be the unique parametrization that allows hyperparameter 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 large neural networks and thus provide empirical motivation for the theoretical study of the infinite-width μP limit.

B Which Hyperparameters Can Be Transferred? (Continued)

B.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 D.1 on ResNet.

Transferable Hyperparameters In Table 2, we summarize which hyperparameters 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 hyperparameters when the need for regularization is not paramount, e.g., during large scale language model pretraining. Parameter Multipliers are not well-known hyperparameters, 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 hyperparameters 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. Indeed, intuitively, regularization needs to be applied more heavily in larger models, so naturally we do not expect the same regularization hyperparameters to stay constant across model sizes.

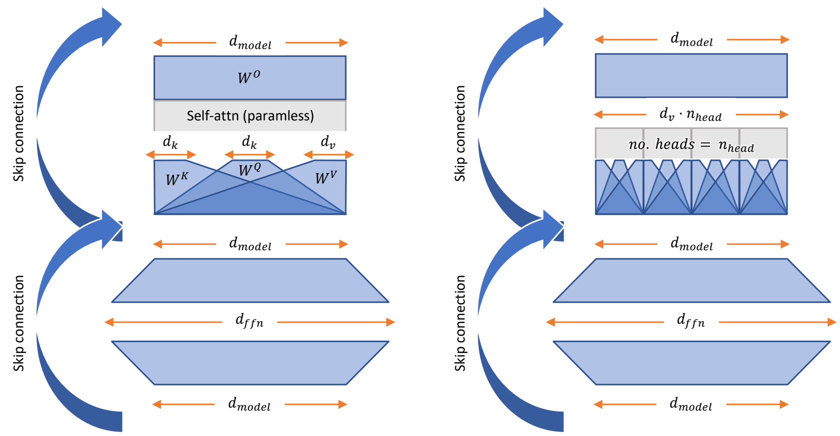
To the best of our knowledge, there is no strict separation between hyperparameters that regularize and those that don't. However, conventional wisdom tells us that there exists a spectrum of how much regularizing effect a hyperparameter has. For example, dropout probability and weight decay are among 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 hyperparameters 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 [6, 14], improvements along any of these *scale* dimensions lead to apparently sustainable gain in performance; as a result, we are primarily interested in transferring other hyperparameters *across* these dimensions that define scale, rather than finding the optimal scale.[17] This category of hyperparameters is particularly crucial as one can speedup training by

[16]But it certainly does not implement other parts of our parametrization, like Adam learning rate scaling or the output multiplier.

[17]In particular, we are not fixing the total training FLOPs when we scale, which requires understanding the tradeoff of different scale hyperparameters. 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.



(a) Single-head attention (b) Multi-head attention

Figure 6: 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}.

downsizing in one or multiple such dimensions. Indeed, it's very common for practitioners to implicitly transfer hyperparameters 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 hyperparameter 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 hyperparameters 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 [29, 31] where our transferability result of learning rate across batch size recapitulates [21]. [18] See Appendix A.2 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 D.1.

B.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. 6) can all be construed as measures of width. We briefly discuss these here, with more theoretical justification in Appendix G.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 as the width of every layer still tends to infinity, we still obtain approximately the same limit[19], so the hyperparameter transfer remains theoretically justified.

See Fig. 7 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 $d_{model} = d_{head} \times n_{heads}$. 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_{heads} as the width, or increasing both simultaneously. This allows our technique to handle many popular models, including GPT-3 [6], which scale up by fixing d_{head} and increasing n_{head}. See Fig. 8 for an empirical validation on Wikitext-2.

[18] There's also a literature on the proper initialization for training deep networks effectively (e.g. [5, 11, 19, 28, 41, 42, 45]), but they do not study the *transferability* per se. See Appendix A.2

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

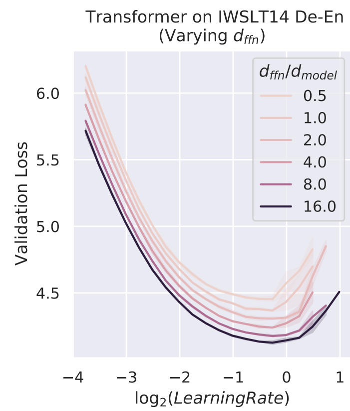


Figure 7: Learning rate landscape in μP is stable even if we vary d_{ffn} by a factor of 32, fixing d_{model}.

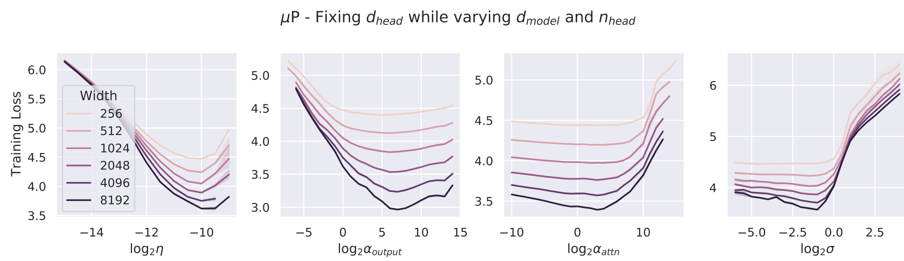


Figure 8: Hyperparameters transfer across width when we fix d_{head} and vary d_{model} and n_{head}. $\alpha_{output}, \alpha_{attn}$ are multipliers for output and key weights, and σ is initialization standard deviation.

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 hyperparameter landscape can be quite noisy. Keeping d_k relatively large while shrinking all other dimensions solves this problem, while still obtaining significant speedup.

C Experimental Details

C.1 IWSLT

IWSLT14 De-En is a well-known machine translation benchmark. We use a Transformer implemented in `fairseq` [22] with a default $d_{model} = {}^1\!/_4 d_{ffn} = 512$ and $d_k = d_q = d_v = {}^{d_{model}}\!/_{n_{heads}} = 128$ (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 $z \in \{-3, -2, -1, ..., 8\}$

C.2 WMT

We scale up to WMT14 En-De using the large Transformer from [35], with a $d_{model} = {}^1\!/_4 d_{ffn} = 1024$ and $d_q = d_k = d_v = {}^{d_{model}}\!/_{n_{heads}} = 64$. 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_{heads} from 16 to 4. This reduces the total parameter count from 211M to 15M. We then perform the hyperparameter 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 $z \in \{-3, -2, -1, ..., 8\}$

C.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 hyperparameters 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 hyperparameters 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 hyperparameters in SP BERT Large.

Finetuning Procedure and Hyperparameters We hand-pick the finetuning hyperparameters after training the full-sized model. As regularization is an essential ingredient in successful finetuning, we do not perform hyperparameter transfer (at least the suite of techniques presented in this work) (see Table 1). We focus on MNLI [38] and QQP, which are two representative tasks from GLUE [36]. Following [20], we used Adam [15] 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.

D Additional Experiments

D.1 Experiments on ResNets

D.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 hyperparameter 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. 9, on the left.

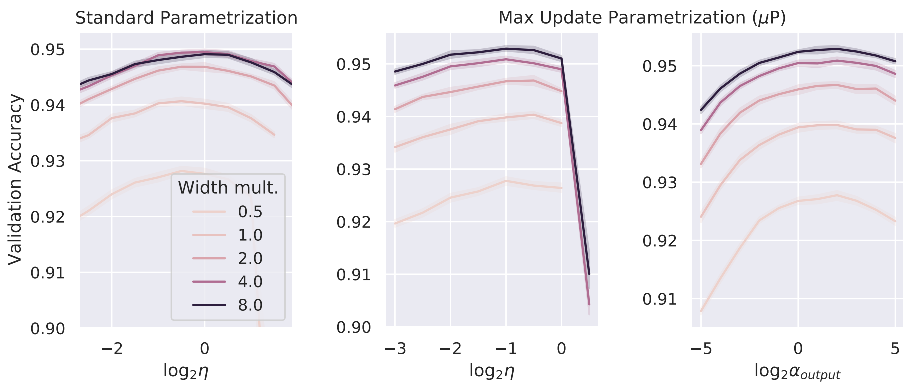


Figure 9: ResNet on CIFAR-10 for different widths (compared to a base network). On the **left**, the widest network SP underperforms; on the **right**, the μP network has a more consistent hyperparameter landscape and performs better. Both networks are tuned at the smallest width for the hyperparameter (η or α_{output}) not in the x-axis.

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. 9, 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.[20] 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 7, where μP outperforms SP by $0.43\% \pm .001\%$.

Table 7: 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 hyperparameters are different as the models are parametrized to be identical at 1x width.[20]

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%**

D.1.2 Wide ResNet on ImageNet

Setup For this case we use Wide-Resnet, or WRN [44], a ResNet variant with more channels per layer, to further showcase hyperparameter transfer 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 [44].

Hyperparameter Transfer We start with a proxy model with a width multiplier of 0.125 and tune several hyperparameters 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 hyperparameters used by [1] for ResNet-50; while not expected to be competitive for WRN, they represent a reasonable starting point for our experiment.

[20] 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.

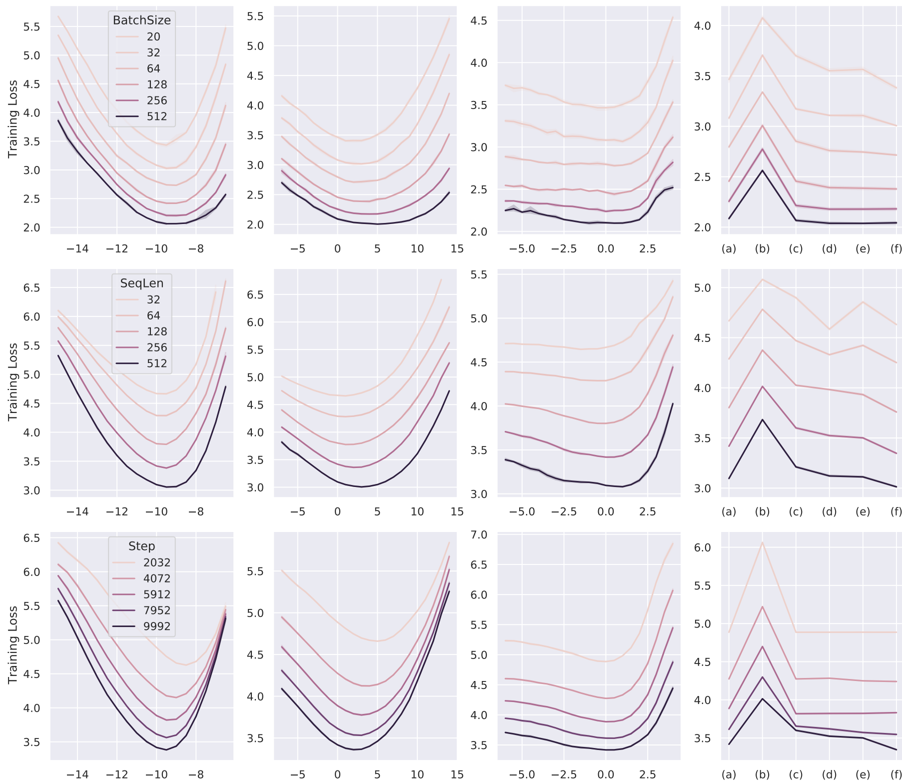


Figure 10: **Empirical validation of Hyperparameter Transfer across Batch Size, Sequence Length, and Training Time on pre-LN Transformers.** Same setting as Fig. 4. Despite some shift, the optimal hyperparameters are roughly stable when transferring from batch size 32, sequence length 128, and 5000 training steps.

We randomly sample 64 hyperparameter 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 hyperparameters we just selected. The result is shown in Table 8, where μP outperforms SP by 0.41% in terms of top-1 validation accuracy.

Table 8: 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%**

D.2 Experiments on Transformers

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

See Fig. 10.

D.3 Post-Layernorm Transformers

Fig. 11 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.

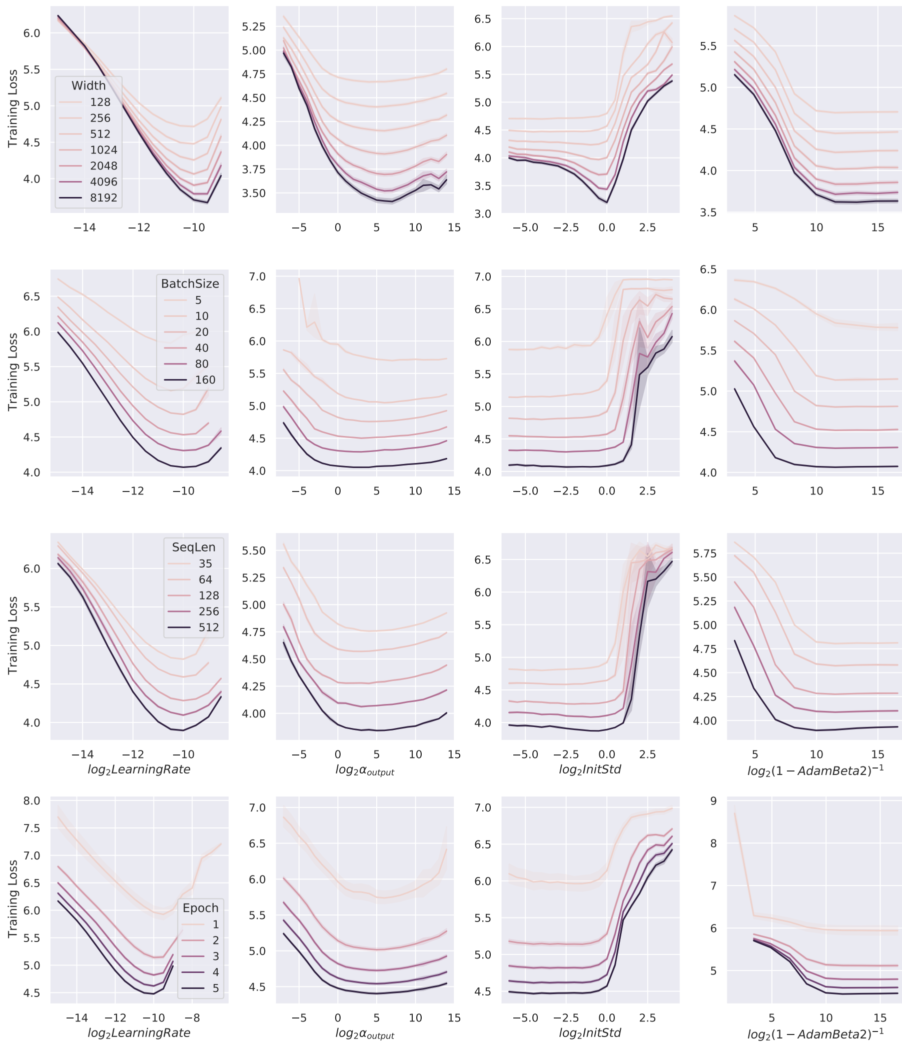


Figure 11: **Empirical validation of Hyperparameter Transfer for Post-LN Transformers.** Same setting as Fig. 4.

D.3.1 Hyperparameter Instability of SP Transformers

Fig. 12 and Fig. 13 show the hyperparameter instability inherent in SP Transformers.

E Implementing Hyperparameter Transfer in a Jiffy

As we have shown, one can enable hyperparameter transfer 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 model `base_width`.

The latter goal, which we call *parametrization backward compatibility*, ensures that any code base works exactly as before when model width equals `base_width`, 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 width differs from `base_width`, the model behavior necessarily changes so that hyperparameters can be transferred. Most commonly, the user should set the `base_width` to be the width of the target model

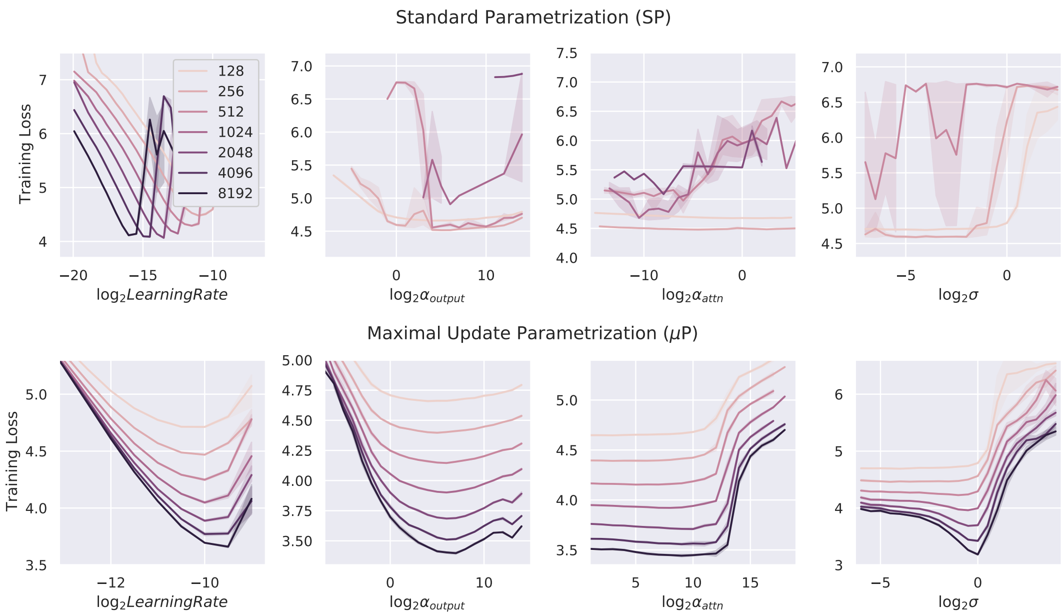


Figure 12: Post-layernorm Transformer with SP and μP on Wikitext-2. We sweep one hyperparameter across width (d_{model}) at a time while keeping the rest fixed; we also scale d_{head} linearly with d_{model} and fixing n_{heads}. $\alpha_{output}, \alpha_{attn}$ are multipliers for output and key weights, and σ is initialization standard deviation. This yields unstable result for SP, as expected, where missing points/curves represent divergence; in μP, the optimal hyperparameter choices stabilize as width increases.

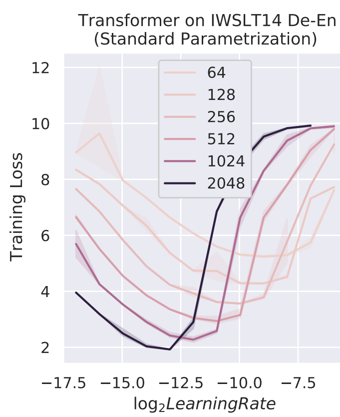


Figure 13: Learning rate landscape is highly unstable under standard parametrization in IWSLT.

(e.g. BERT-large or T5-large). Then one can tune a proxy model with e.g. $width = \texttt{base\_width}/4$ to obtain the optimal hyperparameters for the target model. In addition, if one wishes to scale up further e.g. $width = 4 \times \texttt{base\_width}$, then these hyperparameters remain optimal. Of course, depth, batch size, and sequence lengths can be scaled up and down as well according to Fig. 10.

The `MUP` Package We provide our tool as a Python package called `MUP` designed to work with PyTorch. For the most generic use case, where one scales the widths of all layers at once, the transition to μP boils down to 3 steps:

1. Replace the input and the output layers with counterparts in `MUP.layer` and specify a `base_width` for both;
2. Ensure all other layers are initialized with `fan_in` initialization;[21]

[21] This is the default behavior for Pytorch `nn.Linear` layers, but some code bases then manually overrides this initialization e.g. with constant init.

3. Replace the optimizer (e.g., Adam) with the counterpart (e.g., `MuAdam`) in `MUP.optim`.

What Happens in the `MUP` Package The `MuLayers` take care of the parametrization for input/output layers and label them for the optimizer. The `MuOptimizer` reads the `base_width` from the input layer and calculates the learning rate scaling for all parameters. For example, `MuAdam` scales the learning rate for the input/output layers like $\Theta(1/\sqrt{width})$, one-dimensional parameters (e.g., gains and biases) like $\Theta(1)$, and other parameters like $\Theta(1/width)$. It might seem odd that the optimizer "micromanages" the learning rate for gains and biases; this design choice is motivated by minimizing the required code change by the user, as the alternative is to replace every layer that has gains or biases.

F Width-Related Training Issues are Hyperparameter Issues

Large transformers are famously fickle to train [18, 26]. This is especially true in low-precision floating point formats such as `float16` that is required to train enormous models today. Throughout our investigations, we find that, with the hyperparameters transferred from the proxy model (which should be close to optimal), we do not observe such instability. This suggests that the training instability problems could in fact be hyperparameter problems, not necessarily structural issues associated with the architecture or optimizer (which work just fine when tuned properly).[22] We hypothesize that

The common manifestations of training instability are caused by practitioners naively transferring learning rate and other hyperparameters tuned on a small transformer.

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-transferring the instability-inducing hyperparameters from a large transformer to a small one and replicating the training instability.* This is shown in Fig. 14.

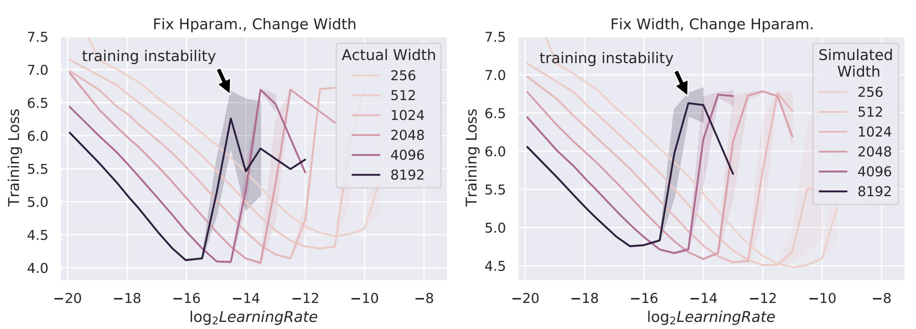


Figure 14: **Replicating training instability on a small transformer by *reverse-transferring* hyperparameters.** These experiments concern 2-layer Transformers in Standard Parametrization (SP) on Wikitext-2, trained with Adam, where width is defined as $d_{model} = d_{ffn}$. (Left) LR-vs-loss for wider 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[23]; in particular, a learning rate leads to instability in a wide model iff it does so when transferred back to a narrow model.

[22] Of course, it is still worthwhile to research layers or architectures that are more insensitive to poor hyperparameter choices. But with our results, good hyperparameters are obtained much more easily, so this issue is now much less urgent.

[23] 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.

Table 9: 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	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta(n)$

G An Intuitive Introduction to the Theory of Maximal Update Parametrization

In what follows, we seek to describe useful intuitions and rule of thumbs that would be helpful to practitioners and empirical researchers alike in figuring out what is the right neural network parametrization. Readers needing rigorous justification can generally find it in [39, 40].

G.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. (6)). 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.* (6)*).* 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 $\Theta(1)$.[24] 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.[25] This is summarized in Table 9, 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.

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

G.1.1 Preparation for the Derivations

By the results of [40], each (pre-)activation vector and its gradient vector in a multi-layer perceptron have approximately iid coordinates in the large width limit,[26] and something similar can be said for more advanced networks such as ResNet and Transformers [27]. In particular, to each such vector v, we can associate a random variable Z^v that represents the coordinate distribution of v. If vector u is correlated with v, then Z^u will also be correlated with Z^v, and $\lim_{n\to\infty} v^\top u/n = \mathbb{E}\, Z^u Z^v$.

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

[25] 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.

[26] 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.

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

G.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 so, we can form the outer product

$$A \stackrel{\text{def}}{=} u \otimes v/n = uv^\top/n, \tag{5}$$

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 $Z^{Ax} \stackrel{\text{def}}{=} Z^u\, \mathbb{E}\, Z^v Z^x$. 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).[28]

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

$$A_{\alpha\beta} = \psi(u^1_\alpha, \ldots, u^k_\alpha, v^1_\beta, \ldots, v^k_\beta), \quad \text{for some } \psi : \mathbb{R}^{2k} \to \mathbb{R} \text{ and vectors } u^i, v^j \in \mathbb{R}^n. \tag{6}$$

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. (5)) already takes this form, with $\psi(u^1_\alpha, \ldots, u^k_\alpha, v^1_\beta, \ldots, v^k_\beta) = n^{-1}(u^1_\alpha v^1_\beta + \cdots + u^k_\alpha v^k_\beta)$, so Eq. (6) 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).[29] If these unnormalized gradients are the outer products $u^1 \otimes v^1, \ldots, u^k \otimes v^k$, then the update has coordinates

$$(\mu/\sigma)_{\alpha\beta} = \psi(u^1_\alpha, \ldots, u^k_\alpha, v^1_\beta, \ldots, v^k_\beta) \stackrel{\text{def}}{=} \sum_i \gamma_i u^i_\alpha v^i_\beta \Big/ \sqrt{\sum_i \omega_i (u^i_\alpha v^i_\beta)^2}, \tag{7}$$

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. (6), 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. (7), 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,

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

Here we made the obvious definition

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

[28] 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.

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

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

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

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 $\Theta(1)$.

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

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

G.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.

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 $\mathbb{R}^n$.[30] 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 [39] for more details.

G.2 Deriving μP for Any Architecture

Armed with the insight from the last section, we now outline the key steps to derive μP for any architecture. In practice, μP of [40] implies the following desiderata

Desiderata G.1. At any time during training

1. Every (pre)activation vector in a network should have $\Theta(1)$-sized coordinates[31]
2. Neural network output should also be $\Theta(1)$.
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

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

[31] 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

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.

Finally, desideratum 3 means that 1) we are doing "maximal feature learning" [40] 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 [40], and this property theoretically guarantees hyperparameter transfer across width (for sufficiently large width). However, for the sake of reaching a broader audience, we will focus more on the intuitive derivations from the desiderata rather than on this formal aspect.

Below, we 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.

G.2.1 μP Derivation From the Desiderata

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 $Wx = \Theta(1)$. We can initialize W with $\Theta(1/n)$ coordinates and scale its (per-layer) LR so that ΔW has $\Theta(1/n)$ coordinates as . However, in order to use the same SGD learning rate for all layers, we instead equivalently 1) reparametrize $W = \frac{1}{\sqrt{n}}w$ with trainable w, 2) initialize w with $\Theta(1/\sqrt{n})$ coordinates, and 3) use $\Theta(1)$ learning rate (of w) for SGD. For Adam, however, we cannot use the same learning rate for every layer, so we set the Adam learning rate of w to be $\Theta(1/\sqrt{n})$.

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 9, 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 desiderata 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. However, in order to use the same SGD learning rate for all layers, we instead 1) reparametrize $W = \sqrt{n}w$ with trainable w, 2) initialize w with $\Theta(1/\sqrt{n})$ coordinates, and 3) use $\Theta(1)$ for SGD learning rate. Again, for Adam LR, we have to use a per-layer learning rate (of w), for which we take $\Theta(1/\sqrt{n})$.

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.

It turns out the correct choice, as shown in Table 3, is to replace n with fan-in for the input layers and with fan-out for the output layers. 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.

G.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 9.
2. For SGD, the gradient of $\mathbb{R}^{n \times n}$ weight has $\Theta(1/\sqrt{n})$ coordinates because the last layer has no $1/\sqrt{n}$ factor, 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 and thus does not allow hyperparameter transfer for the same reason that NTP below does not.

Neural Tangent Parametrization (NTP) We have concrete examples, e.g. Word2Vec in [40], where the NTK limit has trivial performance — so hyperparameters have no effect at all — vastly outperformed by finite-width networks — where hyperparameters matter. More importantly, wider does not always do better in NTP, especially in tasks where feature learning is crucial [40]. 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 [40], which says that any nontrivial stable "natural parametrization" (formally, "*abc-parametrization*," [40]) 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 hyperparameters?

We argue no. As shown in [40], any other feature learning parametrization differs from μP essentially only in that some parameters are not updated maximally. By [40, 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. Therefore, we cannot hope for the hyperparameter landscape of the limit to reflect the hyperparameter 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 hyperparameter landscape of the μP limit should reflect the hyperparameter landscape of finite-width neural networks.

H Nuances of the Hyperparameter Landscape Convergence Intuition

What converges and what doesn't We want to tune hyperparameters on a small model with width N such that its hyperparameter landscape looks like that of a large model with width $\gg N$. 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 hyperparameter optimum converges, but 2) cannot be so large that the functional dynamics converges. The fact that such N exists, as demonstrated by our experiments, shows that: In some sense, the hyperparameter optimum is a "macroscopic" or "coarse" variable which converges quickly with width, while the neural network function is a very "microscopic" or "fine" detail that converges much more slowly with width.

The same discussion applies to scaling of batch size as well [21].