

## A Details on studying the relationship between gradients and weights in Section 2.1

The training details used are summarised in Table 6. In Figure 5, we illustrate how the relationship between weights and gradients evolves for three representative components: the positional embedding weights, the value weight matrix in layer 6, and the weight matrix in the final classification layer, corresponding to progressively deeper layers in the network.

As shown in the figure, parameters with large gradients typically correspond to those with small weight magnitudes (except in the case of the final classification layer as shown in the appendix). There are several possible explanations for the distinct behaviour observed in the final classifier layer. First, the classifier is newly initialised from scratch, rather than being inherited from pretrained weights. Second, it needs to adapt to the task-specific label space. Consequently, in our algorithm, we exclude the final layer, along with normalisation layers, from selective updates. In contrast, for the other layers, the strong correlation between gradient magnitude and weight magnitude remains significant. This finding also supports the observation made in [25], where models were found to frequently update parameters with smaller weight magnitudes. However, in their work, this phenomenon was not further explored; instead, they adhered to prior practice by using gradients as the primary criterion for parameter importance.

Table 6: Hyperparameters for studying the gradient-weight relationships.

Setting	LR	WD	Batch size	Epoch	Label Smooth
CoLA (BERT-base, FT)	7e−5	0.0	32	5	—
CIFAR10 (ViT, FT)	3e−3	0.1	128	300	0.1
CIFAR10 (ViT, Scratch)	3e−3	0.1	128	300	0.1

## B Overparameterization leads to stronger correlation

We provide additional visualisation of gradient-weights distribution in FT ViT-Tiny and ViT-Large models on CIFAR10 in Figure 6.

## C Additional theoretical statements and motivation

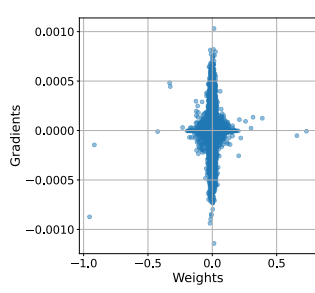
We provide more details regarding the two layers neural network setup. For pretraining, the data is generated from a similar structured teacher network  $f_{\text{teacher}}$  with  $n = 2$  neurons and input dimension  $d = 2$ . Furthermore, all neurons are first initialized  $a \sim \text{Uni}(\{-1, 1\})$  i.e. i.i.d. Rademacher random variables and  $w_{i,j} \sim N(0, 1)$  for all  $i, j \in [k, d]$  and normalized over the input dimension. We initialize the dense network with  $k = 20$  and the so-called COB initialization based on the rich regime in [5]. All weights are initialized with  $\mathcal{N}(0, 1/n)$  and we ensure that  $a_i = -a_{i+10}$  for  $i \in [10]$ . We train for  $T = 10000$  steps with learning rate  $\eta = 2$ . This training gives us the pretrained network  $f_{\text{pre}}$ . For the fine tuning we generate additional neurons in the same way as the teacher neuron. We train for  $T = 10000$  steps with learning rate  $\eta = 1$ . A slightly smaller learning rate has been chosen to ensure convergence for the large gradient setup.

**Gradient flow** The gradient flow of a two layer neural network for training one neuron is given by

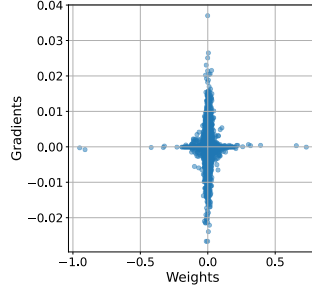
$$\begin{cases} da_t = -\frac{1}{n} \sum_i \sigma(w_t^T x_i) (a_t \sigma(w_t^T x_i) - y_i) dt, & a_0 = a_{\text{init}} \\ dw_t = -\frac{1}{n} \sum_i a_t x_i \mathbb{I}_{w_t^T x_i > 0} (a_t \sigma(w_t^T x_i) - y_i) dt, & w_0 = w_{\text{init}}, \end{cases}$$

As highlighted the study of nano gradient flow can be reduced to studying the case of training one neuron where the labels are generated by a teacher neuron  $f_{\text{extra}}$ . This is under the assumption that the chosen neuron is not contributing to the representation.

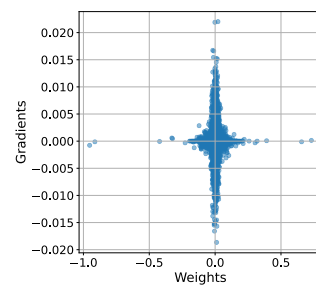
**Task difficulty measure for two-layer network** The theoretical measure of how difficult our post training task is captured by the distance in the function space  $L_2(\mathbb{R}^d, p)$  between a reference task  $\tilde{f}$



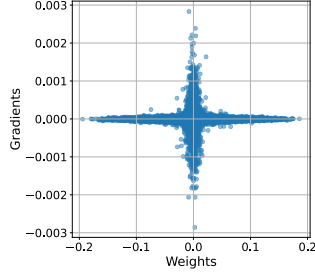
(a) positional embedding weight, early of FT.



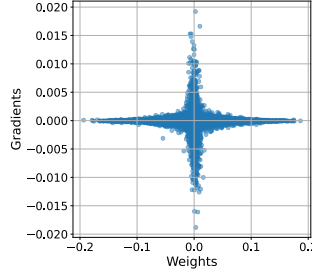
(b) positional embedding weight, middle of FT.



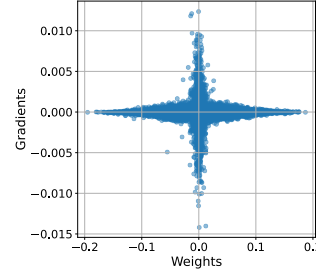
(c) positional embedding weight, end of FT.



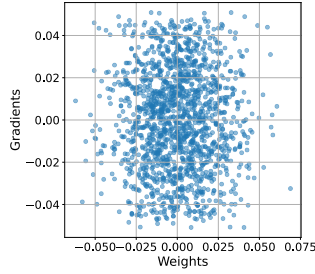
(d) value weight in layer 6, early of FT.



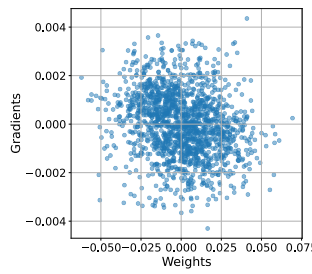
(e) value weight in layer 6, middle of FT.



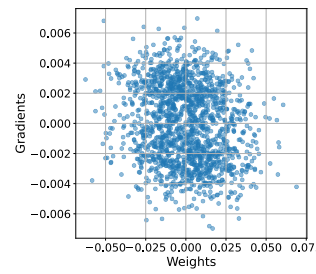
(f) value weight in layer 6, end of FT.



(g) classifier weight, early of FT.



(h) classifier weight middle of FT.



(i) classifier weight, end of FT.

Figure 5: The dynamic of the relationship between gradients and weights during finetuning Bert-base on COLA. The x-axis represents the magnitude of the weights, while the y-axis represents the magnitude of the gradients. From left to right, the subfigures correspond to the early, middle, and late stages of finetuning. From top to bottom, the subfigures represent progressively deeper layers in the network.

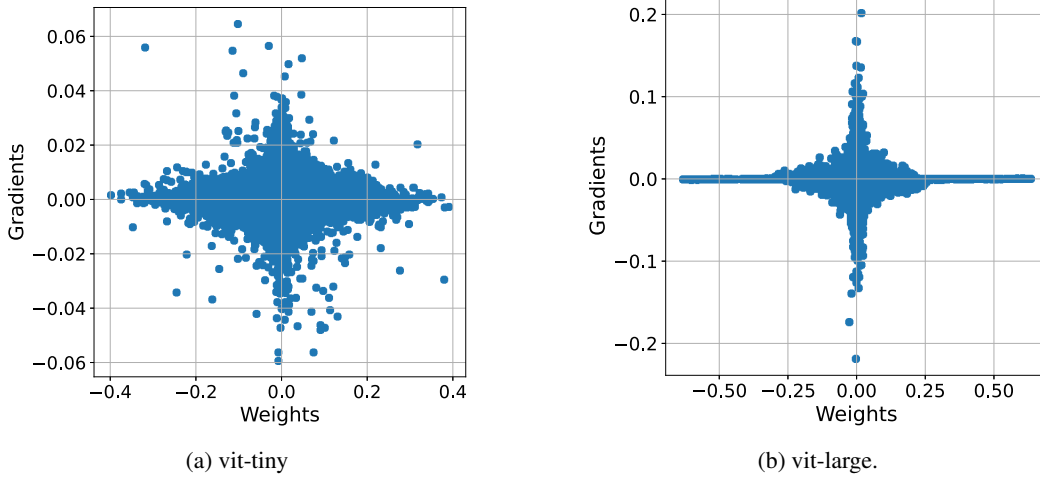


Figure 6: Gradients and weights distribution of first layer in FT ViT-Tiny and ViT-Large at early stage of FT. Overparameterisation leads to more hyperbolic relationship.

863 and final representation  $f$ . Concretely the measure is defined as

$$A_{\text{task}}(f, \tilde{f}) := \|f - \tilde{f}\|_{L_2(\mathbb{R}^d, p)}^2 = \int_D \left| \sum_i a_i \sigma(w_i^T x) - \sum_i \tilde{a}_i \sigma(\tilde{w}_i^T x) \right|^2 dp(x)$$

864 where  $p$  is a probability measure on the data space. The distance measure can be approximated by the  
865 use of the empirical measure and or an upper bound solely depending on the weight space. In the  
866 main text we assumed that we learned the representation  $f_{\text{pre}}$  and that the finetuning task is given by  
867  $f_{\text{fit}} = f_{\text{pre}} + f_{\text{extra}}$

868 **Lemma C.1** Denote the weights  $a_i \in \mathbb{R}$  and  $w_i \in \mathbb{R}^d$  for  $i \in [n]$  of  $f_{\text{extra}}$  and  $p$  is a Gaussian with  
869 mean  $\mu = 0$  and covariance matrix  $\Sigma = I$ . Then

$$A_{\text{task}}(f_{\text{fit}}, f_{\text{pre}}) \leq \sum_i |a_i|^2 + \|w_i\|^2$$

870 Proof. (1) Apply the triangle inequality neuron wise (since we have learned the teacher representation.  
871 (2) Gaussian integral calculations for a ReLU activation. (3) Apply Cauchy-Schwarz inequality to  
872 each  $a_i |w_i|$ .

873 Lemma C.1 substantiates the use of the  $\ell_2$  distance in our toy example. Note the bound is tight in the  
874 balanced case.

Table 7: Average test loss on finetuning task and distance from pretrained initialization over 10 seeds. Small weights leads to better generalization and move less from the original representation.

Algorithm	Test Loss	$\ell_2$ Distance
Small Weights	$0.0094 \pm 0.012$	$0.027 \pm 0.0040$
Large Gradient	$0.017 \pm 0.015$	$0.057 \pm 0.032$

## 875 Single neuron fine turning

876 **More neurons fine turning** We repeat the same experiment as in the main text but with an  
877 additional neuron. In Table 8 we observe that the variance for the distance by selecting the large  
878 gradients becomes high. This is in line what is observed in Figure 7c where we learn a complete new  
879 representation.

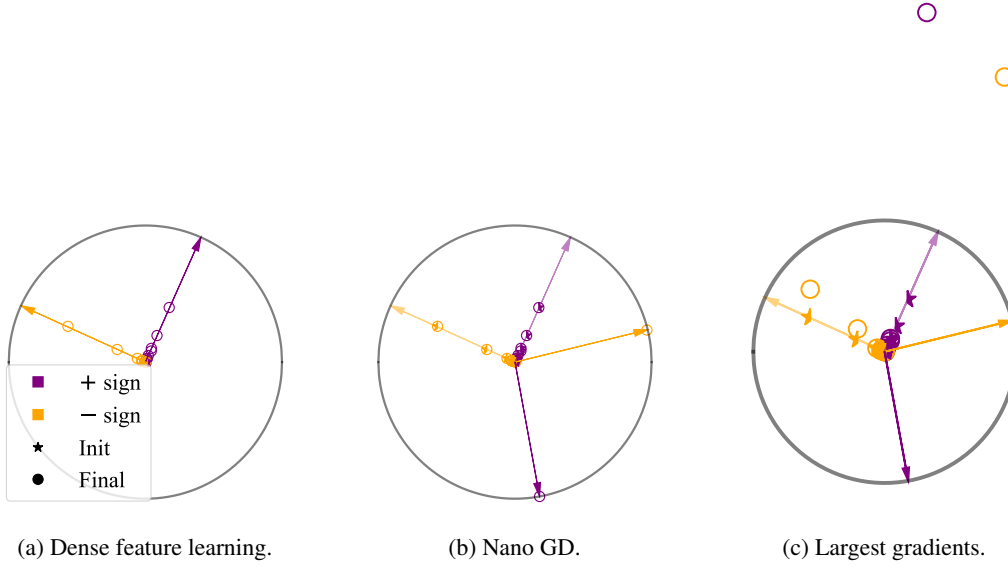


Figure 7: nano gradient descent provably prevents catastrophic forgetting. (a) Two layer student network learns teach networks representation. (b) Nano gradient descent keeps the original representation while learning the two extra neuron. (c) The largest gradients can lead to learning completely different representations when the task transferability is high.

Table 8: Average test loss on finetuning task and distance from pretrained initialization over 10 seeds. Small weights leads to better generalization and move less from the original representation even when a more difficult representation needs to be learned i.e. less transferable tasks.

Algorithm	Test Loss	$\ell_2$ Distance
Small Weights	$0.038 \pm 0.040$	$0.042 \pm 0.0077$
Large Gradient	$0.063 \pm 0.033$	$0.097 \pm 0.071$

## D Ablation study

### D.1 Small vs. large vs. random weights

For all configurations, we use a learning rate of  $9 \times 10^{-5}$ , weight decay of 0.0, a batch size of 32, 5 training epochs, and a fixed random seed of 42. The mask density is initialized at  $k_0 = 0.01$ , the mask update interval is set to  $m = 131$ , and the density scheduler is disabled for this experiment. The training loss and evaluation metrics over steps are shown in Figure 8.

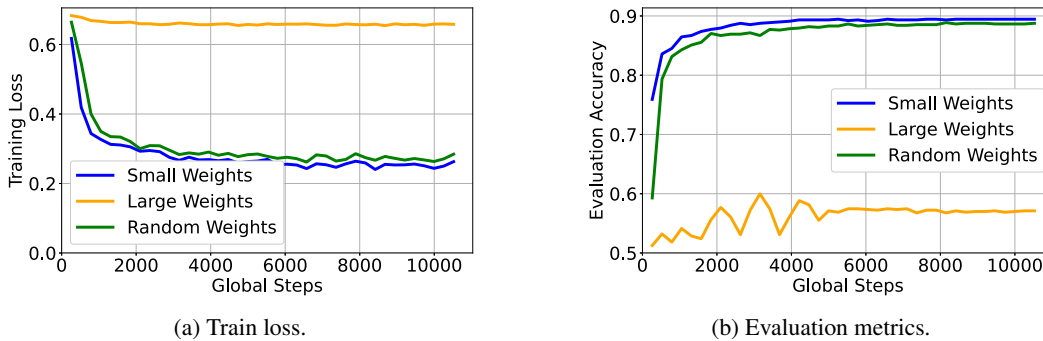


Figure 8: Ablation study comparing three masking strategies in NANOADAM: small-magnitude weights, large-magnitude weights, and random weights. Small-weight masking achieves the best training loss and evaluation performance under the same gradient density.

## 885 D.2 Small weights vs. large gradients

886 **Study in NLP domain** We conduct an ablation study comparing two parameter selection strategies:  
 887 (1) selecting parameters with the smallest absolute weight magnitudes, and (2) selecting parameters  
 888 with the largest absolute gradient magnitudes. All experimental configurations follow the setup  
 889 described in Appendix D.1, with the exception that the learning rate is separately tuned for each  
 890 strategy to their best performance. The optimal learning rate is  $1 \times 10^{-3}$  for small weights and  
 891  $3 \times 10^{-4}$  for large gradients. The corresponding training loss and generalization performance are  
 visualized in Figure 9.

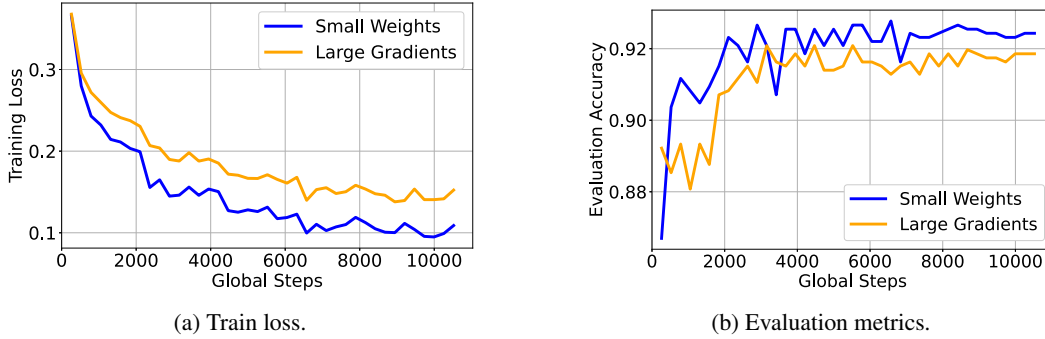


Figure 9: Ablation study comparing two selection criteria: small weights and large gradients. Small weight is better at generalization and convergence.

892

893 **Study in vision domain** We also provide an ablation study on FT CV tasks, where we compare  
 894 NANOADAM under two different masking strategies: (1) Large gradients: selecting parameters with  
 895 the largest absolute gradient magnitudes. (2) Small weights: selecting parameters with the smallest  
 896 absolute weight magnitudes. Specifically, we finetune the ViT-Large model on the Flowers102  
 897 dataset, using the same hyperparameter settings detailed in Table 9. We initialize the mask density  
 898 at  $k_0 = 0.001$ , and turn the density scheduler off. We set the mask update interval to  $m = 100$ .  
 899 The resulting training loss and evaluation accuracy are presented in Figure 10. Note that in this  
 900 experiment, we do not perform learning rate search for both strategies. Instead, we keep the same  
 901 learning rate for both cases. Although the final performance is similar with both strategies, small  
 weights achieve faster convergence.

Table 9: Hyperparameters for fully finetuning ViT-Large on Flowers102.

LR	weight decay	batch size	epoch	seed
$3e-3$	0.0	128	10	42

902

## 903 D.3 Dynamic mask vs. static mask

904 To evaluate the effectiveness of dynamic masking, we conduct experiments similar to the previous  
 905 setup, with the key difference being the masking strategy used in NANOADAM. Specifically, for  
 906 all configurations, we use a learning rate of  $9 \times 10^{-5}$ , weight decay of 0.0, a batch size of 32, 5  
 907 training epochs, and a fixed random seed of 42. The mask density is initialized at  $k_0 = 0.01$ , and  
 908 the density scheduler is disabled for this experiment. We compare two approaches: (1) Dynamic  
 909 masking, where the mask is updated every  $m = 131$  steps; and (2) Static masking, where a fixed  
 910 mask from the beginning is applied throughout the entire training process. The resulting training  
 911 loss and evaluation accuracy are shown in Figure 11. While the static mask achieves very similar  
 912 evaluation performance during the initial phase of training, the dynamic mask continues to improve  
 913 and ultimately surpasses the static strategy in both evaluation accuracy and final training loss. These

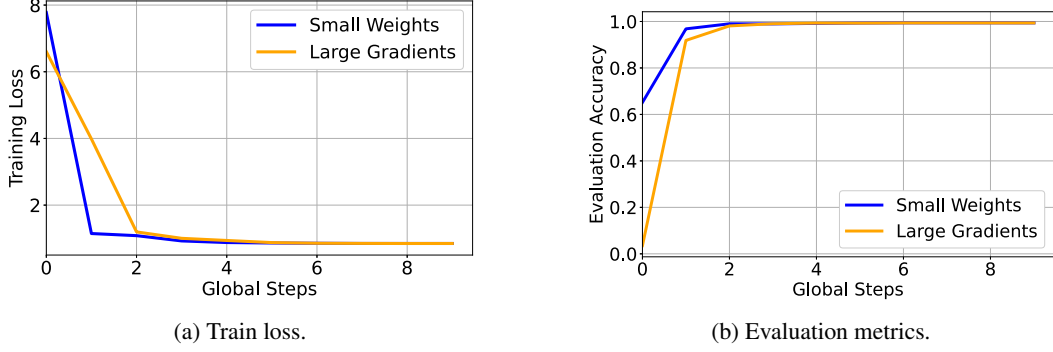


Figure 10: Ablation study on ViT-Large finetuned on the Flowers102 dataset comparing two selection criteria: small weights and large gradients. Small weight is better at generalization and convergence.

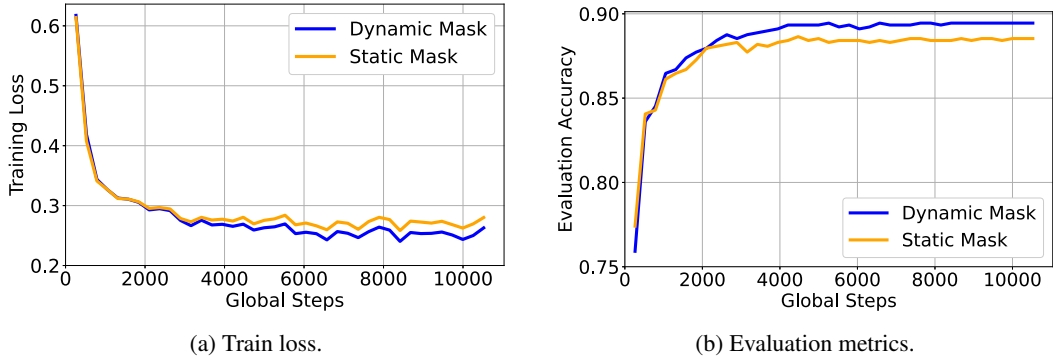


Figure 11: Ablation study on BERT-base finetuned on the SST2 task comparing two masking strategies in NANOADAM: dynamic mask and static mask. Dynamic mask achieves the best performance.

914 results indicate that dynamic masking allows the model to adapt more effectively throughout training,  
 915 leading to better convergence and generalization.

## 916 E Theoretical memory footprint comparison

917 We now compare the theoretical memory footprint of NANOADAM with other optimizers, including  
 918 AdamW, AdamW-8bit, and MicroAdam, focusing specifically on the optimizer state memory. Fol-  
 919 lowing the setup in [22], we assume the total number of model parameters is denoted by  $S$ . We use  
 920 the LLaMA-2 7B model as a concrete example to illustrate memory consumption.

921 **NanoAdam.** NANOADAM stores the optimizer states  $m$  and  $v$  (first- and second-order momentums)  
 922 only for the selected subset of parameters in `bf16` format, with each requiring 2 Bytes. Given a  
 923 gradient density  $k$ , the number of updated parameters is  $kS$ . Therefore, the total memory required  
 924 for the momentums is:  $2 (\text{states}) \times 2 (\text{Bytes}) \times kS = 4kS$  Bytes. Additionally, NANOADAM stores  
 925 the mask  $I$  indicating the indices of selected parameters. Simply using 64-bit integers (`long`), each  
 926 index takes 8 Bytes, resulting in:  $8 \times kS = 8kS$  Bytes. Alternatively, if we store indices in `int16`  
 927 format (2 Bytes), as done in MicroAdam, the total memory reduces to:  $4kS + 2kS = 6kS$  Bytes. For  
 928 finetuning LLaMA-2 7B ( $S = 6.275 \text{ B}$ ) with  $k = 0.01$ , NANOADAM requires  $6 \times 0.01S = 0.3765$   
 929 GB.

930 **AdamW.** Stores both  $m$  and  $v$  for all parameters in `bf16`, requiring  $2 \times 2 \times S = 4S$  Bytes.  
 931 For LLaMA-2 7B, this equals  $4S = 25.1$  GB.

932 **AdamW-8bit.** Stores  $m$  and  $v$  in 8-bit precision, needing  $2 \times 1 \times S = 2S$  Bytes, which corresponds  
 933 to  $2S = 12.55$  GB for LLaMA-2 7B.

934 **MicroAdam.** Stores 4-bit quantized error (0.5*S* Bytes) and a sliding window of size *m* for both  
 935 parameter indices in `int16` and values in `bf16`. Each step in the window stores *kS* parameters,  
 936 leading to  $0.5S + 4mkS$  Bytes. For  $k = 0.01$ ,  $m = 10$ , MicroAdam requires  $0.5S + 4 \times 10 \times 0.01S =$   
 937 5.65 GB to finetune LLaMA-2 7B.

## 938 F Details and more results for finetuning on GLUE benchmark

### 939 F.1 Hyperparameters for finetuning on GLUE benchmark

940 We largely follow the hyperparameter settings established by [22] for finetuning on various GLUE  
 941 tasks. Specifically, we finetune for 5 epochs with a per-device batch size of 8, a fixed random seed  
 942 of 42, and no weight decay. Unless otherwise stated, we perform grid search over the learning rate  
 943 values  $\{1e-6, 3e-6, 5e-6, 7e-6, 1e-5, 3e-5, 5e-5, 7e-5\}$  for all optimizers and models.

944 **GaLore.** For GaLore, we set the low-rank approximation rank to  $r = 256$  and vary the SVD  
 945 update interval  $T \in \{20, 200\}$ . In contrast to the original GaLore implementation, which tunes both  
 946 the scale and learning rate, we fix the scale to 1 and augment the learning rate search space with  
 947  $\{1e-4, 3e-4, 5e-4, 7e-4\}$ .

948 **MicroAdam.** For MicroAdam, we use a sliding window of  $m = 10$  gradients and a sparsity level  
 949 of  $k = 1\%$ , resulting in an effective gradient sparsity of  $mk = 10\%$ . The quantization bucket size is  
 950 set to 64.

951 **Adam and Adam-8bit.** All general hyperparameter settings mentioned above are directly applied  
 952 to both Adam and Adam-8bit baselines.

953 **NanoAdam.** For NANOADAM, we set the initial update density to  $k_0 = 10\%$  and linearly decay it  
 954 to 4% by the end of training. As training small weights typically requires a larger learning rate, we  
 955 search over a wider range:  $[5e-6, 3e-4]$ . Additional hyperparameters specific to NANOADAM are  
 summarized in Table 10.

Table 10: Hyperparameters for NANOADAM across tasks.

task	COLA	SST2	MRPC	STSB	QQP	MNLI	QNLI
mask interval	6	52	7	13	711	306	81
density interval	33	263	14	27	1423	1533	409

956

### 957 F.2 Additional results for finetuning on GLUE benchmark

958 The peak memory usage and running time are reported in Table. 11 and 12.

### 959 F.3 Training Dynamics for finetuning NLP task

960 We also present the training dynamics observed in an NLP finetuning task. Specifically, we analyze  
 961 the training loss and generalization performance of various optimizers when finetuning a BERT-base  
 962 model on the QNLI task from the GLUE benchmark. For each optimizer, we perform hyperparameter  
 963 tuning to determine the optimal learning rate. The experiment settings are the same as those described  
 964 in Appendix F.1. The best learning rates for each method are summarized in Table. 13.

965 The resulting training loss and evaluation accuracy over time are shown in Figure 12a and 12b. As  
 966 expected, in the early training steps, full finetuning with AdamW achieves the lowest training loss,  
 967 since it updates the entire parameter set. However, at later stages, NANOADAM surpasses AdamW in  
 968 both training loss and generalization. This is because updating only small-magnitude weights initially  
 969 has minimal impact on the model output—dominated by large weights—but gradually exerts greater  
 970 influence as training progresses.

971 In terms of generalization performance, AdamW performs better in the initial steps but is soon over-  
 972 taken by NANOADAM, which consistently achieves higher accuracy in the later stages. Furthermore,

Table 11: Memory usage (GB) on GLUE dataset.

Model	Method	COLA	SST2	MRPC	STSB	QQP	MNLI	QNLI	AVG.
BERT -BASE	Microadam	3.64	3.63	3.64	3.64	3.81	3.75	3.79	3.70
	NANOADAM	3.58	3.60	3.59	3.57	3.60	3.60	3.59	<b>3.59</b>
	Galore	4.06	4.05	4.06	4.06	4.05	4.05	4.05	4.06
	AdamW-8b	3.72	3.72	3.72	3.72	3.72	3.72	3.72	3.72
	AdamW	3.94	3.94	3.95	3.95	3.94	3.93	3.95	3.94
BERT -LARGE	Microadam	5.56	5.53	5.52	5.54	5.54	5.53	5.54	5.54
	NANOADAM	5.21	5.24	5.15	5.19	5.23	5.22	5.19	<b>5.20</b>
	Galore	6.12	5.60	6.11	6.10	5.90	5.89	5.90	5.94
	AdamW-8b	5.61	5.83	5.62	5.61	5.61	5.62	5.60	5.64
	AdamW	6.45	6.52	6.47	6.47	6.52	6.47	6.46	6.48
OPT -1.3B	Microadam	13.20	13.15	13.19	13.19	13.19	13.19	13.20	13.19
	NANOADAM	11.33	11.76	11.41	12.04	11.66	11.77	11.57	<b>11.65</b>
	Galore	14.18	14.18	14.39	14.26	14.18	14.18	14.17	14.22
	AdamW-8b	13.07	13.08	13.08	13.08	13.08	13.08	13.08	13.08
	AdamW	18.18	18.16	18.16	18.16	18.17	18.17	18.16	18.16

Table 12: Training time (minutes) on GLUE dataset.

Model	Method	COLA	SST2	MRPC	STSB	QQP	MNLI	QNLI	AVG.
BERT -BASE	Microadam	3.25	17.79	1.78	6.26	98.79	103.42	28.41	37.10
	NANOADAM	2.99	16.57	1.46	2.09	91.35	94.05	26.72	33.60
	Galore	2.01	12.16	0.86	1.51	72.65	69.99	19.08	25.47
	AdamW-8b	1.40	8.04	0.66	1.09	57.95	45.96	15.81	18.70
	AdamW	1.13	7.38	0.55	0.88	43.63	39.14	10.93	14.80
BERT -LARGE	Microadam	7.26	37.30	2.85	4.47	170.67	175.73	54.80	64.73
	NANOADAM	4.36	28.99	2.33	3.23	157.40	164.44	47.97	58.39
	Galore	4.36	26.72	1.87	3.27	162.00	160.34	44.90	57.64
	AdamW-8b	3.88	18.85	1.08	1.79	96.64	93.20	25.38	34.40
	AdamW	2.04	12.91	0.93	1.56	78.37	78.07	21.16	27.86
OPT -1.3B	Microadam	9.17	46.14	5.55	7.10	238.46	253.18	70.35	89.99
	NANOADAM	4.95	37.12	2.54	4.23	186.92	196.63	53.74	69.45
	Galore	12.50	84.24	4.85	8.37	451.76	463.27	129.09	164.87
	AdamW-8b	3.91	29.39	1.74	2.88	160.03	158.81	44.31	57.30
	AdamW	3.38	27.73	4.27	2.53	138.25	135.58	38.54	50.04

across all training steps, NANOADAM outperforms MicroAdam, demonstrating its superior learning dynamics.

We also visualize the dynamics of the ratio between the number of parameters updated at least once and the total number of parameters during finetuning on the CoLA task using BERT-base. The results are shown in Figure 12c. Note that optimizers such as GaLore, AdamW, and AdamW-8bit update all parameters by design; thus, their curves are omitted for clarity. As shown, MicroAdam eventually updates over 90% of the parameters, whereas NANOADAM keeps more than 80% of parameters untouched throughout training.

Table 13: learning rate for various optimiser on finetuning Bert-base on QNLI.

optimiser	AdamW	AdamW-8b	GaLore	MicroAdam	NANOADAM
LR	$7e-5$	$7e-5$	$1e-4$	$4e-5$	$1.1e-4$

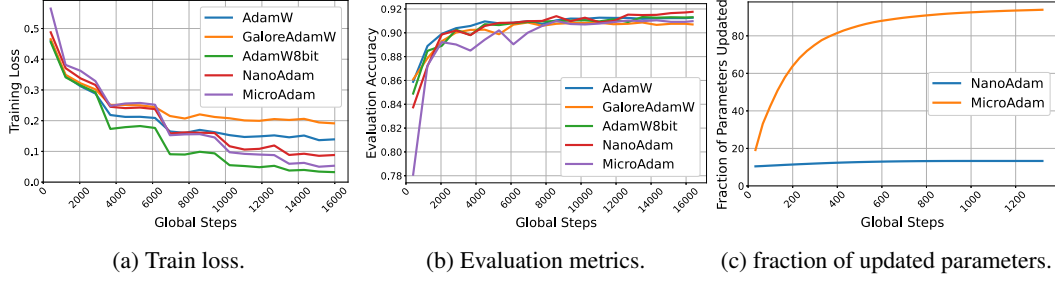


Figure 12: Dynamics of NLP FT.

## G Details and more results for experiments on CV Tasks

### G.1 Details of Experiments on CV Tasks

**ViT-Large** The detailed training configurations for the ViT-Large model are summarized in Tables 14–15, including both common and optimizer-specific hyperparameters. Task1 is CIFAR10 and task2 is Flowers102.

Table 14: Common hyperparameters used for finetuning ViT-Large.

Batch Size	Seed	Weight Decay	LR Scheduler	Label Smoothing
128	42	0.0	CosineAnnealingLR	0.1
Epochs Task 1	Epochs Task 2	$\beta$	$\epsilon$	–
5	5	(0.9, 0.999)	$1 \times 10^{-8}$	–

Table 15: Optimizer-specific hyperparameters for ViT-Large.

Optimizer	LR Task1	LR Task2	$k / k_0$	Dynamic Density / $m$	Mask Interval
NANOADAM	1e-3	2e-3	0.1%	off	100
MicroAdam	1e-4	1e-3	0.1%	$m = 10$	–
AdamW	1e-4	1e-4	–	–	–

**ResNet101** The experimental settings for ResNet101 are summarized in Tables 16–17. These include common training hyperparameters and optimizer-specific configurations. Task1 is CIFAR10 and task2 is Flowers102.

Table 16: Common hyperparameters for ResNet101.

Batch Size	Seed	Weight Decay	LR Scheduler	Label Smoothing
128	42	0.0	None	0.0
Epochs Task1	Epochs Task2	$\beta$	$\epsilon$	–
30	30	(0.9, 0.999)	1e-8	–

**ResNet18** For ResNet18, we use the same common settings as in Table 16, while the optimizer-specific hyperparameters for ResNet18 are summarised in Table. 18. Task1 is CIFAR10 and task2 is Flowers102.

Table 17: Optimizer-specific hyperparameters for ResNet101.

Optimizer	LR Task1	LR Task2	$k / k_0$	Dynamic Density / $m$	Mask Interval
NANOADAM	1e-2	7e-3	1%	off	100
MicroAdam	1e-3	5e-3	0.1%	$m = 10$	–
AdamW	1e-3	1e-3	–	–	–

Table 18: Optimizer-specific hyperparameters for ResNet18.

Optimizer	LR Task1	LR Task2	$k / k_0$	Dynamic Density / $m$	Mask Interval
NANOADAM	9e-3	7e-3	1%	off	100
MicroAdam	1e-3	5e-3	1%	$m = 10$	–
AdamW	1e-3	1e-3	–	–	–

## G.2 Dynamics of catastrophic forgetting

The generalisation performance of various optimisers over different tasks are shown in Figure 13- 15, while the experiment settings are reported in Appendix G.1.

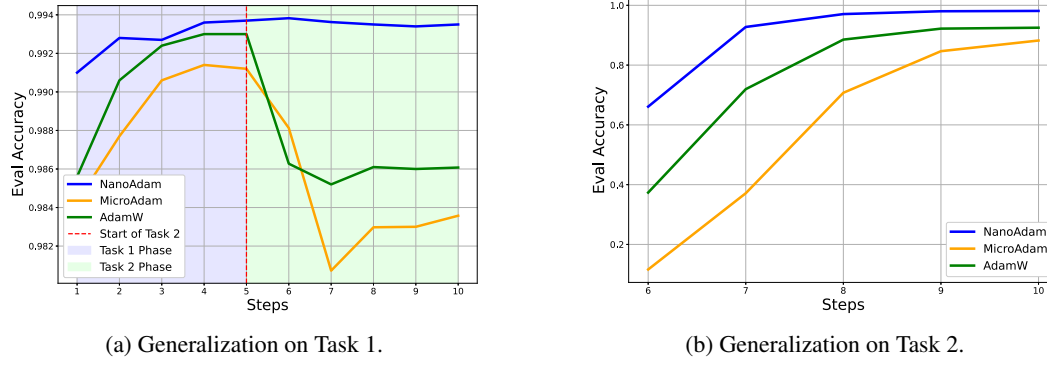


Figure 13: Catastrophic forgetting with ViT-Large.

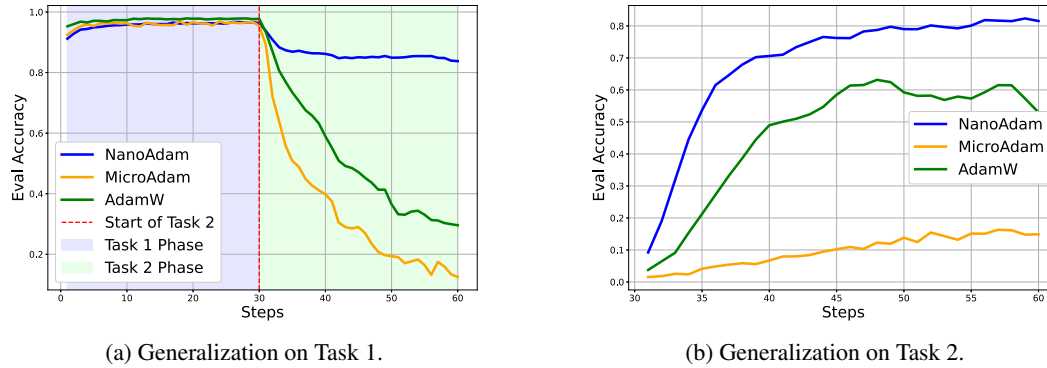


Figure 14: Catastrophic forgetting analysis on ResNet101. (a) Accuracy on Task 1 drops significantly after switching to FT on Task 2 for AdamW and MicroAdam, while NANOADAM retains performance. (b) NANOADAM also achieves better adaptation on Task 2.

## G.3 Parameter shift visualisation

To provide finer-grained insights, we visualize the layer-wise differences between the parameters of the pretrained ViT-Large model and those after continual learning on CIFAR10 and Flowers102, pre-

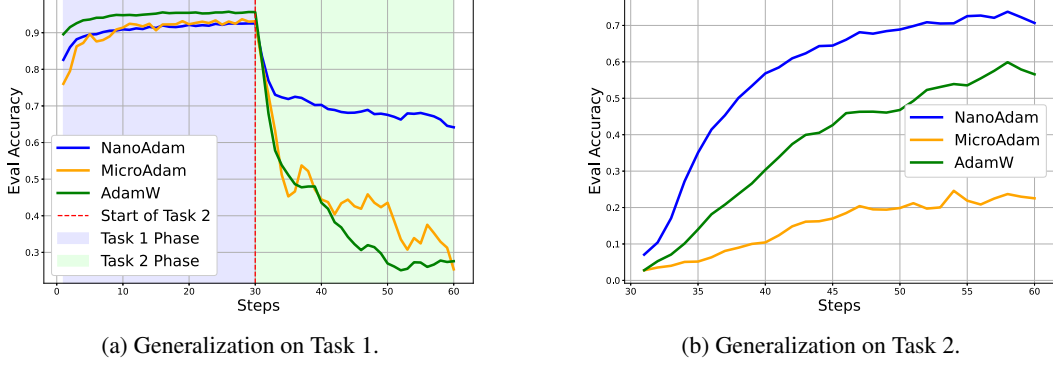


Figure 15: Catastrophic forgetting with ResNet18.

998 sented as heatmaps in Figure 16. The experimental setup follows the details reported in Appendix G.1.  
 999 Notably, NANOADAM induces minimal drift in the attention weights (QKV), highlighting its stability  
 in preserving critical features during continual learning.

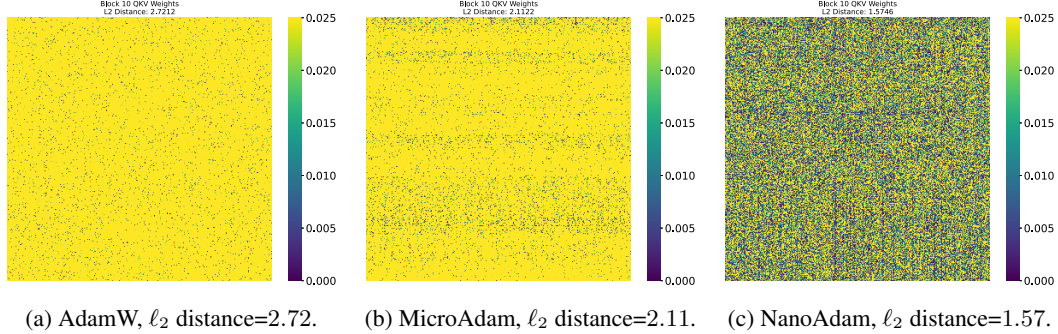


Figure 16: Parameter shift (qkv block) in Layer 10 after continual finetuning on CIFAR10 and Flowers102.

#### G.4 Larger learning rate

We further investigate whether NANOADAM enables stable optimization under larger learning rates. We finetune ViT-Large (pretrained on ImageNet) on CIFAR-10 using NANOADAM, MicroAdam, and AdamW, representing finetuning of small weights, large gradients, and all weights respectively.

Using the same hyperparameter settings as in Tables 14–15, we vary the learning rate in  $\{1e-5, 1e-4, 1e-3\}$ . The resulting performance is shown in Figure 17. We observe that both AdamW and MicroAdam perform best at  $1e-4$  and degrade significantly at  $1e-3$ . In contrast, NANOADAM can benefit from the larger learning rate, achieving its best performance at  $1e-3$ . This highlights its stability and effectiveness in aggressive optimization regimes.

#### G.5 Effective learning rate

Effective Learning Rate (ELR) quantifies the actual rescaling applied to parameter updates during optimization. In adaptive optimizers such as Adam and its variants, the update rule incorporates element-wise adaptation based on the historical statistics of gradients. For a parameter  $w$  at step  $t$ , the update is given by:

$$w_{t+1} = w_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

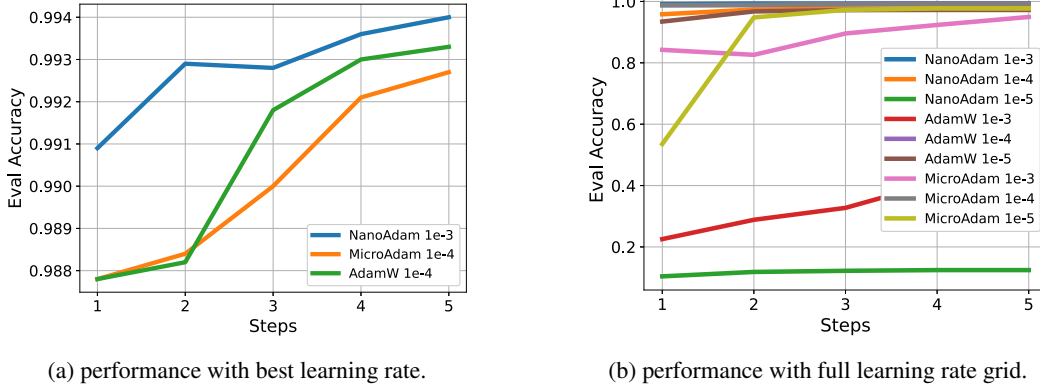


Figure 17: Small weights allow large learning rate.

Here,  $\eta$  denotes the global learning rate;  $\hat{m}_t$  and  $\hat{v}_t$  represent the bias-corrected first and second moment estimates, respectively; and  $\epsilon$  is a small constant added for numerical stability. The *effective learning rate* is thus defined as:

$$\text{ELR} = \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}$$

Since ELR is computed per-parameter and evolves over time, its magnitude provides insight into how aggressively each parameter is being updated. For sparse optimizers such as NANOADAM and MicroAdam, which selectively update a subset of parameters, we compute ELR only over the actively updated parameters. The final reported metric is the average ELR across these selected parameters. We present the ELR dynamics of NANOADAM, MicroAdam, and AdamW during fine-tuning on both vision and language tasks.

For the computer vision task, we fine-tune the ViT-Large model (pretrained on ImageNet) on CIFAR10 using various optimizers. The experimental settings follow those described in Tables 14–15. The ELR trends for different optimizers are depicted in Figure 18. For the NLP task, we fine-tune the BERT-base model on the SST-2 dataset from the GLUE benchmark. The experimental details are provided in Appendix F.1. As shown in the results, NANOADAM enables a more aggressive effective learning rate compared to other methods.

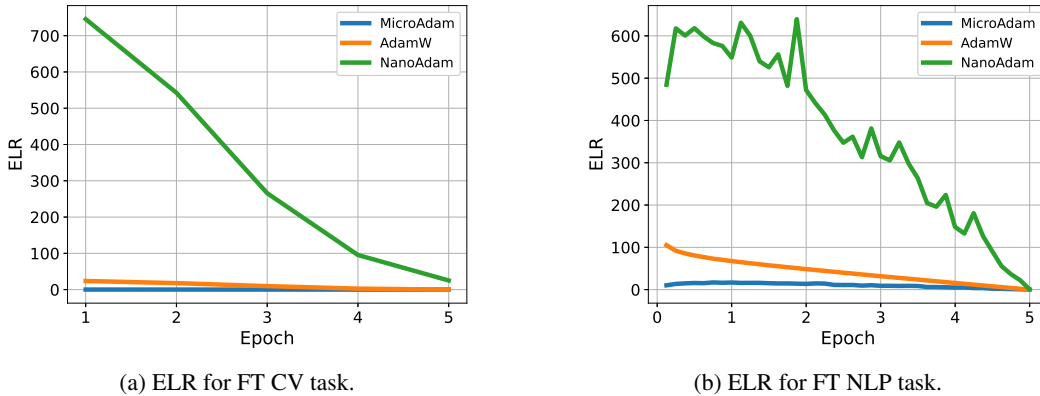


Figure 18: Comparison of effective learning rate in FT CV and NLP task.